Universidad de Alcalá
Escuela Politécnica Superior

**Grado en Ingeniería Electrónica de Comunicaciones**

# Trabajo Fin de Grado

Desarrollo de aplicaciones para microntroladores con Amazon FreeRTOS (a:FreeRTOS)



**Autor:** Sergio Santamaría Quevedo

**Director:** José Manuel Villadangos Carrizo

2021

# UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

**Grado en Ingeniería Electrónica de Comunicaciones**

**Trabajo Fin de Grado**

**Desarrollo de aplicaciones para microntroladores con Amazon FreeRTOS (a:FreeRTOS)**

Autor: Sergio Santamaría Quevedo

Director: José Manuel Villadangos Carrizo

**Tribunal:**

**Presidenta:** Ana Jiménez Marín

**Vocal 1º:** César Mataix Gómez

**Vocal 2º:** José Mauel Villadangos Carrizo

Calificación: .....

Fecha: .....

# Resumen

El objetivo del proyecto es la realización, ejecución y puesta en práctica de ciertos procedimientos con el fin de comunicar un dispositivo basado en microcontrolador con un servicio en *la nube* como es el que ofrece Amazon, para poder crear una red propia o sistema de *el Internet de las Cosas* (IoT). Tras la explicación teórica de cada una de las partes, se realizarán varios experimentos a modo de pasos intermedios, que buscan la comprensión de ideas y conceptos. Tras ellos, se partirá de un ejemplo para el desarrollo de un firmware que consta de una medida de temperatura y humedad y su envío a dicha nube. Finalmente, gracias a las librerías que ofrece AWS se accederá a la información enviada por el dispositivo y representarla, de forma local, en una interfaz gráfica escrita en Python.

**Palabras clave:** Amazon Web Services (AWS), FreeRTOS, microcontrolador STM32, Internet of Things (IoT), Python.

# Abstract

The present report is created to document the realisation and execution of the procedures to be able to communicate a microcontroller-based device with a *cloud* service. A service such as the one offered by Amazon, in order to create its own network or system for the *Internet of Things* (IoT). After the theoretical explanation of each of the parts, several experiments will be carried out as intermediate steps, aimed at understanding ideas and concepts. After these, an example for the development of a firmware consisting of a temperature and humidity measurement and its sending to the cloud will be used as a starting point. Finally, thanks to the libraries offered by AWS, the information sent by the device will be accessed and represented locally in a graphical interface written in Python.

**Keywords:** Amazon Web Services (AWS), FreeRTOS, STM32 microcontroller, Internet of Things (IoT), Python.

# Resumen extendido

Amazon tiene su sección *Amazon Web Services* (AWS) donde ofrece servicios de todo tipo en su nube. Uno de ellos es *el Internet de las Cosas* (IoT). El propósito de este trabajo es mostrar los procedimientos necesarios para que, con una tarjeta basada en un microcontrolador de 32 bit (un dispositivo de relativa poca potencia), se puedan medir dos magnitudes (temperatura y humedad) y enviarlas a la nube de *Amazon Web Sevices Internet of Things* para su –presumible– posterior proceso.

Para poder realizarlo, en primer lugar, se describirá de forma resumida cómo moverse en el entorno AWS, cosa que no es sencilla pues es bastante poco intuitivo. Hay que crear una cuenta raíz y configurar muchas opciones para poder tener un mínimo de usabilidad. Posteriormente, se pasará a la parte del IoT para realizar más configuraciones que resultarán en unas comunicaciones seguras entre los objetos (se verá qué es un objeto en ese entorno) y AWS IoT.

Por otro lado se explicará qué es el protocolo MQTT, ampliamente implantado como protocolo de comunicaciones entre dispositivos de baja potencia de cálculo y baja energía. Se realizarán unas cuantas pruebas externas para asentar conceptos antes de ver cómo hacerlo en AWS IoT. Del mismo modo que el protocolo MQTT, también se expondrá qué es el formato de texto `.json` que, junto al protocolo anterior, forman un estándar *de facto* en las comunicaciones IoT.

En cuanto a la parte hardware, se usarán dos tarjetas de desarrollo: una ESP32, basada en un microcontrolador de 32 bit de Espressif y la Discovery kit for IoT node, basada en un STM32L4 con múltiples sensores integrados. Con el ESP32 se harán unas pruebas muy interesantes que demostrarán que poder conectar a internet –y a AWS– no es exclusivo de dispositivos más avanzados. Con el firmware que trae preinstalado el STM32 mostrará sus capacidades. La tarjeta seleccionada para el desarrollo final será la STM32L4 Discovery kit for IoT node convirtiéndose así en el *edge device*.

El *edge device* (que así se llaman los dispositivos de relativa poca potencia de cálculo basados en microcontrolador para la adquisición de datos) se programará usando el «servicio Amazon FreeRTOS» que no es más que FreeRTOS con unas librerías que añade Amazon para facilitar las comunicaciones con el núcleo de AWS IoT y algún extra que también simplifica el desarrollo.

Tras ejecutar un ejemplo (que ofrece Amazon desde su servicio FreeRTOS) sobre la tarjeta basada en STM32, se analizará la forma de adaptarlo a los requerimientos del presente TFG, esto es medir temperatura y humedad, y enviarlo en un formato definido a la nube AWS IoT. Se analizará el esquema de funcionamiento del ejemplo, sus partes y en qué punto se pueden realizar las adaptaciones para transformarlo en un *firmware* que cumpla con las espectativas del TFG.

Con posterioridad, se realizará una interfaz en Python para PC y, de esa forma, ver gráficamente la evolución de los parámetros medidos. La interfaz se conectará al mismo nodo en el que se publican los datos, se procesarán (mínimamente) y se mostrarán de una forma más clara, intuitiva y visual que viendo números y caracteres en una terminal. Además de añadir esa mayor comprensión visual, con este *script* se comprueba que no es estrictamente necesario usar los servicios de AWS, se pueden realizar aplicaciones *ad-hoc* y conectarse a la nube.

Finalmente se bosquejarán los pasos para poder almacenar los datos adquiridos y enviados a AWS IoT en una base de datos de AWS. Cuando un *edge device* envía información, esta es remitida a los dispositivos que están a la escucha (suscritos), pero en ningún momento se guarda esa información por parte de AWS IoT. Para poder guardar las mediciones, se hace necesario configurar unas pocas políticas y reglas, además de crear una base de datos para recibir los registros. De esta forma es posible recuperarlos *a posteriori* para realizar análisis, estudio y estadísticas.

### Nota del Autor

En el momento en el que se aceptó el anteproyecto, Amazon ofrecía su «a:FreeRTOS» como una herramienta para los *edge divices* que facilitaba enormemente la conexión de dichos dispositivos con la nube de Amazon. En las fechas en las que se escribe esta memoria, Amazon ha hecho desaparecer el término, pero sólo el término, pues sigue ofreciendo la herramienta ahora transformada en un servicio, el servicio Amazon FreeRTOS. Es un cambio de nomenclatura, si se quiere es un “ascenso de categoría” de mera herramienta a servicio, pero sigue siendo lo mismo. El autor ha tratado de seguir los pasos de Amazon cambiando el término –sólo el término– en el interior de la memoria por el de «servicio Amazon FreeRTOS». La gran excepción es el título del TFG que, por razones conocidas, no se puede cambiar. Si se ha escapado alguna entrada sin cambiar, pido disculpas por la confusión terminológica.

# Índice general

# Índice de figuras

# Índice de tablas

# Índice de listados de código fuente

# Capítulo 1

# Introducción

Este trabajo mostrará cómo conectar dispositivos basados en microcontrolador –con una potencia de cálculo moderada– a la nube de *Amazon Web Services* (en adelante AWS) dentro del marco de dispositivos *Internet of Things* (en adelante IoT).

La finalidad de las redes IoT es muy diversa: adquisición de datos meteorológicos, control de flotas de vehículos, control de accesos de una fábrica con varias entradas y/o edificios, control de las condiciones climatológicas (temperatura y humedad) de algunas o todas las secciones de dichos edificios... en definitiva, se trata de una red de dispositivos con los que poder obtener una telemetría. El IoT se basa precisamente en eso, en conectar a una red una serie de dispositivos de diferente naturaleza para que puedan intercambiar información entre ellos.

Si bien estas redes pueden montarse de forma local, la alternativa a la conexión a la nube añade una flexibilidad muy alta en cuanto a, por ejemplo, la escalabilidad, pues es más sencillo hacer crecer la cantidad de dispositivos, pero sobre todo añade flexibilidad en la conectividad: los dispositivos de sensado, monitorización y control no tienen por qué estar en la misma red ni en el mismo entorno geográfico.

## 1.1 Motivación y objetivos

Más allá de la propia experiencia de aprender algo nuevo, la finalidad de este TFG es triple:

- Dar una visión de qué elementos conforman un sistema IoT describiendo qué papel juegan y cómo han de configurarse para que todo engrane y funcione.

- Arrojar luz sobre el entorno AWS IoT. Todo (o casi todo) lo realcionado con los servicios AWS son muy potentes y altamente configurables aunque, quizá por eso, son poco intuitivos de configurar para conseguir desarrollar un proyecto.
- Desarrollar un proyecto íntegro con herramientas externas (una interfaz gráfica local) para poder monitorizar la adquisición de datos.

Como objetivos se tendrán los siguientes hitos:

1. Plantear qué es un sistema IoT y sus diferentes componentes, tanto lógicos como físicos.
2. Responder a las necesidades de comunicación: protocolos y formatos de datos.
3. Implementar un pequeño sistema IoT de adquisición y transmisión de medidas, para lo cual hay que conseguir antes:
   (a) Crear una pequeña red IoT local con herramientas de código abierto.
   (b) Mediante esas mismas herramientas de código abierto, establecer una conexión con AWS IoT.
   (c) Conectar dispositivos de diferente naturaleza y programados en distintos IDEs a AWS IoT.
   (d) Programar un microcontrolador (con Amazon FreeRTOS) en un kit de desarrollo empleando las librerías para comunicarse con la plataforma AWS IoT.
   (e) Mostrar los datos adquiridos en tiempo real en forma de gráfico.
   (f) Almacenar los datos adquiridos para que no se pierdan y poder realizar un estudio y procesamiento posterior.

## 1.2 Organización de la memoria

La organización de esta memoria está planteada en cinco grandes capítulos más anexos:

- En este primer capítulo se realiza una breve introducción al Internet de las cosas y se definen los objetivos del presente TFG.
- En el segundo capítulo se expone el marco teórico que abarca a todo el trabajo. Incluído en este, se encuentra el *State of Art* donde se pueden ver las alternativas actuales a la elegida para el desarrollo de este TFG.
- En el tercer capítulo hay un desarrollo real por partes:
  1. Primero se hacen pruebas con distintos dispositivos y *software* creando una pequeña red IoT local.
  2. En segundo lugar se programan y prueban dispositivos para conectarse a AWS IoT.
  3. Finalmente se toma un ejemplo del servicio Amazon FreeRTOS como base, detallando sus partes para conectar un dispositivo a AWS IoT con este sistema operativo integrado en un microcontrolador de 32 bits.
- En el cuarto capítulo se muestran los resultados obtenidos.
- En el último capítulo están las conclusiones y posibles líneas futuras.

# Capítulo 2

# Marco teórico

## 2.1 Introducción

Dado que el presente TFG tiene variedad de conceptos y entornos, el punto de vista teórico se aborda dividiéndolo en:

- *State of Art*. Antes de entrar en la teoría que respecta al TFG, se echa un vistazo a las otras alternativas que se podrían haber tomado.
- Amazon Web Services o AWS, que se subdivide en:
  - Servicio IAM de administración de AWS.
  - Servicio AWS IoT Core.
  - Servicio Amazon FreeRTOS.
  - Servicio AWS Lambda.
  - No-servicio de gráficos.
  - Servicio de bases de datos AWS.
- El protocolo MQTT. Descripción del mismo.
- Los ficheros `.json` y su formato.
- Lenguaje Python.
- Hardware.

## 2.2 State of Art

Se distinguen dos partes claramente distintas que confluyen en este trabajo:

- Los servicios en la nube.
- Los dispositivos físicos que se conectan a la nube.

### 2.2.1 Servicios en la nube

¿Qué son los servicios en la nube? Aunque viene siendo muy popular el término en los últimos tiempos, en realidad se lleva usando desde hace mucho más, puesto que el servicio de correo electrónico se podría catalogar como el primer servicio en la nube de uso masivo. Un servicio en la nube no es más que un programa o aplicación que no están instaladas ni almacenadas de forma local en el PC, si no que son accesibles desde una conexión a internet creando esa ilusión de que están en todos sitios y, a la vez, en ninguno pues da igual el dispositivo que se emplee y el lugar del mundo donde se encuentre el usuario que, con una conexión a internet, puede acceder a él.

Posteriormente también se popularizó el almacenamiento de ficheros en la nube como si de un disco duro virtual se tratase. Quizá fue aquí donde ya se empezó a perfilar el término. Cuando ya se puso en boca de todo el mundo fue a la hora de ofrecer servicios de servidores en la nube. No quiere decir que los servicios anteriores no necesitasen de dichos servidores, es solo que no eran accesibles al usuario. Con estos servicios se ofrecía un servidor conectado a internet, es decir, un ordenador en la nube para poder programarlo según las necesidades pero abstrayéndose por completo de la parte hardware. El usuario no tenía que preocuparse por ella pues sería mantenida por el proveedor que, según el servicio contratado ($\equiv$ la cuota a pagar) sería más o menos potente y escalable.

Más cercano en el tiempo aparecieron los servicios *serverless* que hacían que el usuario se pudiera abstraer aún más del hardware ofreciendo directamente una interfaz *online*. Servicios como bases de datos (por ejemplo) se hicieron muy populares pues ya no había que preocuparse de tener unos servidores locales en las empresas ni el coste de su mantenimiento, ya solamente hay que conectarse a la nube y trabajar haciendo búsquedas, creando nuevos registros... esto también da facilidad y flexibilidad a las empresas/usuarios a expandirse o incluso trasladarse de emplazamiento pues se deja de lado los costes de una migración digital y física de los servidores.

Como es un sector en expansión, van apareciendo nuevos servicios que facilitan aspectos cada vez más concretos del trabajo a las empresas y usuarios, ya hay inclusos servicios de *machine learning*, *big data* y a saber lo que depara el futuro. Todo esto ha dado en un cambio del paradigma en lo que a "*la nube*" se refiere, pasando a denominarse *cloud computing*, que presenta una clasificación más formal de los servicios (extraído de [1] y de [2]):

- Software-as-a-Service (SaaS): Se trata del servicio que más abstrae al usuario del hardware. Son los servicios basados en la web tales como servicio de correo o de almacenamiento como disco duro virtual. En este tipo de servicios apenas hay que configurar nada para poder hacer uso.
- Platform-as-a-Service (PaaS): Ya orientado a desarrolladores, este tipo de servicios ofrece una plataforma sobre la que desarrollar una aplicación. El desarrollador no ha de preocuparse por el mantenimiento o la escalabilidad, éstos son automáticamente realizados por quien ofrece el servicio.

- Infraestructure-as-a-Service (IaaS): Pensado para desarrolladores expertos pues serán ellos quien tengan que gestionar la infraestructura y administrarla por completo. La ventaja es que el desarrollador podrá elegir el hardware que desee y, tras la abstracción del mismo (no hay que olvidar que es un servicio *online*) tendrá que encargarse de todo excepto del mantenimiento hardware. El ejemplo más claro son las máquinas virtuales: el desarrollador será quien elija sobre qué tipo de servidor correrá qué sistema operativo.

En la actualidad hay muchas empresas que ofrecen uno, dos o todos los tipos servicios mencionados y siguen creciendo día a día con nuevos. Eso si, por encima de todas, destacan tres gigantescas: Microsoft, Google y Amazon.

Merece la pena mencionar, aunque solo sea muy por encima, que la contrapartida más pesada en todas estas tecnologías es la **seguridad**. El usuario/empresa pierde control en el momento que sube sus datos a la nube y, si un atacante con malas intenciones accede a dichos datos, puede poner en grandes apuros a la empresa (espionaje industrial) o a la/s persona/s (extorsión, revelado de intimidades).

#### Servicio IoT

Quizá el siguiente paso lógico (dentro de la evolución antes descrita) es ofrecer un servicio de conexión de pequeños dispositivos a la nube que puedan ir aportando datos de diferente naturaleza... aquí nace en Internet de las Cosas (*Internet of Things* o IoT). La cuestión es que, dispositivos de distinta índole, tales como frigoríficos, robots aspiradores, termostatos o máquinas procesadoras de alimentos (en el ámbito doméstico); robots de cadena de montaje y depósitos de líquidos (en el ámbito industrial) se conecten a la nube para que puedan informar con pocos o muchos datos a un "procesador central" de tal forma que éste pueda tomar decisiones para mejorar el rendimiento de un proceso industrial o mejorar el confort de una casa. Un ejemplo claro y sencillo son las impresoras de *renting* de las empresas: las grandes y medianas empresas alquilan impresoras con mantenimiento para no tener que preocuparse por el mismo. Estas impresoras conectadas a internet (IoT) avisan a su servicio técnico y de mantenimiento de una posible avería o de la falta de tinta para que, sin que la empresa contratadora tenga que preocuparse por ello, se reponga y pueda seguir funcionando al 100 %.

Conviene volver a decir que Microsoft, Google y Amazon no son los únicos que ofrecen este servicio, pero son los más potentes actualmente y por ello se hará un repaso de lo que ofrece cada uno apoyándose en [3] y en [4]:

**Google Cloud Platform - Cloud IoT** No hay que presentar al todopoderoso Google. Su nube Iot parece la más sencilla de configurar por su interfaz más intuitiva, pero a la hora de la verdad, no es así. Tiene la capacidad de conectar todas sus aplicaciones gracias a sus grandes motores aunque no tenga mucho sentido conectar unos mensajes IoT a YouTube, por ejemplo. Emplea herramientas tan potentes como *TensorFlow*. No se puede dudar de que su infraestructura es muy potente y estable pero las desventajas son que no tiene tantas capacidades como sus competidores directos y no parece tener tanto soporte como cabría elperar del gigante.

**Microsoft Azure - Azure IoT Suite** Azure es el conjunto de servicios que ofrece Microsoft en la nube. Azure IoT Suite su parte, valga la redundancia, IoT. Tiene la latencia más baja de conexión entre dispositivos y la nube. Tiene una gran cantidad de servicios orientados a desarrollar proyectos IoT con un alto grado de integración entre nubes públicas y privadas. Maneja bien grandes volúmenes de datos y tiene una característica muy buena que no poseen sus competidores: múltiples herramientas para

visualizar datos en tiempo real... que parece que, a la vez es un punto negativo porque no es sencillo configurarlo.

**Amazon Web Services - AWS IoT** Por último, Amazon Web Services IoT o AWS IoT. Tiene muchísimas opciones de IoT y es enlazable (no siempre de forma sencilla) al resto de servicios que ofrece en la nube de forma similar a Google Cloud. Su "consola" (así llaman a la interfaz donde se configura todo) no es muy intuitiva y es complejo de configurar casi cualquier parte. Es el proveedor de servicios con la cuota más alta de mercado durante mucho tiempo y eso se nota en la madurez de sus productos, no en vano grandes compañias internacionales de diversas naturalezas como Coca-Cola y Siemens cuentan con los servicios de AWS. Su lista de servicios no para de crecer, aunque tambien la complejidad de hacerlos funcionar. Pero la característica más deseable[1] es su integración con FreeRTOS para los *edge devices* a través de su servicio Amazon FreeRTOS.

### 2.2.2 Dispositivos IoT

Una ingente cantidad de fabricantes crean y ofrecen otra infinidad de dispositivos capaces de conectarse a la nube con muy diversas capacidades. Se podría partir desde el mismo AWS IoT, que da una lista de "*hardware* certificado que funciona con los servicios de AWS" en [5] donde, en el momento de escribir estas líneas ofrece 53 dispositivos certificados y kits que soportan FreeRTOS y conectan a la nube de AWS IoT; 92 dispositivos que pueden conectarse al Core de AWS IoT; 311 dispositivos que soportan *Greengrass*[2], y así hasta no poder numerarlos todos porque cada poco tiempo se van añadiendo más y pocos salen por obsolescencia.

En definitiva, la práctica totalidad de microcontroladores de 32 bit de la actualidad, junto con unos pocos periféricos como Wifi, Ethernet o Lora, pueden conectarse a la nube. Pero no es exclusiva de los microcontroladores con arquitectura ARM, incluso Arduino, con un microcontrolador de 8 bit se conecta sin muchos problemas.

Añadir que, quizá más importante que las capacidades y/o características del dispositivo/placa de evaluación/kit, es la documentación y soporte que tenga. Normalmente cuanta más documentación, suele tener mejor calidad, pero esto no es una ley y quizá sea un mejor indicativo, las comunidades y foros que haya sobre ellos.

Para el desarrollo de este trabajo se ha elegido uno con muy buenas credenciales y con suficientes periféricos para, si se deja volar la imaginación, hacer muchos experimentos con todos los sensores que tiene en placa, se trata del *STM32 Discovery kit IoT node* pero, como se verá en la misma memoria, uno más sencillo –si cabe– y archiconocido, el *ESP32*, también conecta sin problemas a la red.

## 2.3 Amazon Web Services (AWS)

Antes de profundizar en la parte IoT, conviene tener una idea global de lo que son los (vastos) servicios en la nube de Amazon. Partiendo de la base de que AWS comenzó como un un proveedor IaaS que ha ido evolucionando para ofrecer multitud de PaaS (eso si, sin dejar de ofrecer los IaaS). En [6] Amazon ofrece la puerta de entrada a *Amazon Web Services*, haciendo una presentación de los servicios más destacados. Presenta también a algunos de sus grandes clientes como Netflix, Philips, Enel, General Electrics...

[1] Más deseable para el desarrollo de este proyecto y para el Autor.

[2] *Greengrass* es un *software* que hace de intermediario entre los *edge devices* y la nube de AWS IoT cuando, por ejemplo ésta no está fácilmente disponible, no se pueden conectar todos los dispositivos directamente o incluso puede sustituirla en momentos concretos.

Los servicios se podrían definir como infraestructuras montadas para un propósito concreto, de ahí que aparezca a lo largo de esta sección el término "servicio dedicado". Cada servicio está orientado a un fin concreto. Se trata del concepto *serverless*. Si se mira desde el punto de vista de un negocio, no hay por qué tener y mantener –que es la parte más cara– unos servidores dedicados en una instalación concreta de la empresa, que hay que configurar de forma compleja para que realicen una tarea específica, AWS ya se encarga de ello y sólo hay que preocuparse por los datos que intercambiar con el servicio. Se trata pues, de contratar los servicios que se necesiten en un momento dado y escalarlos a medida de que el volumen de datos/ficheros/cálculos crezca o disminuya.

Si hay que decir que AWS (también) tiene como servicio el concepto clásico de "servidor". Las comillas se deben a que, como norma general, dicho servidor está virtualizado en sus servidores reales. Este servicio se ofrece para cubrir cualquier otra cosa que se quiera hacer y no se pueda hacer con el resto de servicios. AWS lo llama EC2.

Los servicios que ofrece cubren un amplio espectro de las necesidades informáticas que se pueden necesitar:

- Almacenamiento de ficheros. Una de las ideas tradicionales: un disco duro en la nube para almacenar fotos, documentos... etc. AWS lo llama S3 y tiene una forma peculiar de funcionamiento en función no sólo del tamaño, también de la disponibilidad que se quiera tener de los documentos almacenados.
- Bases de datos: Tanto SQL, MongoDB... y una base de datos propia de Amazon que se verá en el apartado 2.3.6.
- El ya mencionado EC2. Servidores en su idea clásica.
- Análisis de datos. Cruzando otros servicios como S3 y bases de datos, puede realizar análisis de datos para poder extraer informaciones.
- Aprendizaje automático/*machine learning*. Múltiples herramientas para el aprendizaje como TensorFlow o Deep Learning.
- AWS Lambda. Un servicio bantante curioso que ejecuta código sin necesidad de estar en ningún sitio concreto, símplemente atiende a eventos y puede interconectar todos los demás servicios.
- Amazon IoT: sobre el que versa este TFG, aunque se trata de un servicio tan inmenso y que crece por momentos que, es imposible de abarcarlo en su totalidad, con lo que se hará una sencilla incursión introductoria.

Estos son sólo un muestra de la ingente cantidad de servicios que ofrece, pero es que, además, va creciendo y ofreciendo cada vez cosas más específicas y con más potencia de cálculo.

### 2.3.1 La capa *free tier*

Si bien es cierto que tiene un gran catálogo de servicios ofertados, muchos de ellos o, al menos, las mejores características de los mismos, son de pago. AWS ofrece una cuenta gratuita para poder probar la plataforma y decidir si merece la pena contratar más servicios o no. Se trata de la cuenta *free tier*. Al darse de alta pedirá datos personales, de facturación y una tarjeta de crédito. Si no se elige ninguna de las opciones de pago, en la tarjeta no se cargará nada[3] **pero** eso si, hay que estar muy pendiente de los servicios que se configuran para usar, pues en muchos de ellos, si se sobrepasa el límite impuesto por la

[3] Quizá un cargo de 1 € o de 1 $ que será devuelto en poco tiempo para comprobar que la tarjeta es válida.

capa gratuíta, se realizarán cargos sin previo aviso. También es cierto que se puede configurar un pequeño servicio de administración para que avise por correo electrónico o por otros métodos que se va a cargar algo en la tarjeta.

Todos los servicios que se emplearán en este TFG entran dentro de la free tier en el momento de su uso. No es óbice para que se conviertan a servicios de pago en un futuro.

### 2.3.2 Administración básica de AWS: el servicio IAM

Cuando se contrata un servicio en la nube siempre hay que pelearse con la parte de administración y AWS, aunque no se salga de la *free tier*, no es menos en ese aspecto. No es divertida, pero hay que gestionar la cuenta. Al crear la cuenta, el usuario y contraseña se comportan como ***root***, es decir, tiene acceso a toda la plataforma, incluidas configuraciones y accesos. Como es un sistema pensado para que puedan acceder muchos usuarios de forma concurrente al mismo o diferentes servicios, al igual que en un sistema linux no es bueno trabajar en el día a día con el usuario root, en AWS tampoco.

Por esa razón, AWS tiene el servicio IAM, que es una forma de crear usuarios limitando y controlando su acceso a las diferentes partes de todo el ecosistema AWS. Una de las utilidades más recomendables (incluso AWS la reconoce como "Mejores prácticas") es la de poder crear un usuario con permisos de administrador pero que no será *root*, así se podrá trabajar con él para el día a día si los riesgos de seguridad que conlleva usar y el tener abierto el super-usuario. Hay más información y tutoriales sobre este aspecto en el propio portal de este servicio en [7].

### 2.3.3 AWS IoT

Cada vez hay más dispositivos que miden magnitudes, detectan niveles o monitorizan procesos tanto en el ámbito industrial (procesos de producción, control de accesos...) como en el ámbito doméstico (teperatura de la casa, detección de consumo eléctrico excesivo...). AWS IoT [8] es una plataforma de servicios que los interconecta entre sí, los conecta a la nube, con bases de datos, con inteligencia artificial para prever posibles fallos o mejoras, calcula estadísticas, etc. Todo ello siempre con un ojo puesto en la escalabilidad, es decir, en cualquier momento se puede hacer crecer la cantidad de dispositivos sin mermar los servicios ofrecidos. Tampoco se olvida de la seguridad pues, como es constante en todo AWS, las comunicaciones siempre irán cifradas, incluso en distintas capas. Dentro de la plataforma, Amazon diferencia en tres bloques los servicios principales de AWS IoT:

- Servicios de control y conectividad:


Figura 2.1: Logos de los servicios de control y conectividad [38]

  - **AWS IoT Core**: Servicio principal que realiza las conexiones entre dispositivos y les permite interactuar transmitiendo los mensajes entre ellos y con otros servicios AWS. Debido a que es uno de los servicios más relacionado con este TFG, se profundizará en secciones más adelante.
  - **AWS IoT Device Defender**: Servicio que monitoriza y audita continuamente las configuraciones de IoT (*sic*) para que no se vuelvan inseguras.
  - **AWS IoT Device Management**: Servicio encargado de facilitar la administración y monitorización remota de dispositivos IoT a gran escala.

- Software de dispositivos:

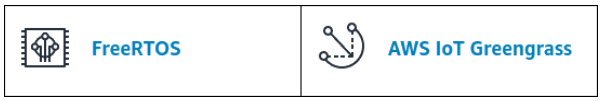


Figura 2.2: Logos de los servicios de software [39]

  - **FreeRTOS**: Servicio que ofrece un sistema operativo para dispositivos de baja potencia de cálculo (microcontroladores) facilitando la implementación y programación de los mismos. Este es el otro servicio más relacionado con este TFG. Al igual que AWS IoT Core, se profundizará en posteriores secciones. También se conoce como servicio Amazon FreeRTOS.
  - **AWS IoT Greengrass**: Servicio/software que ofrece las capacidades de AWS IoT pero de forma local para situaciones o sistemas donde no se pueda acceder de forma fácil a internet y por tanto a AWS.

- Servicios de análisis:


Figura 2.3: Logos de los servicios de análisis [40]

  - **AWS IoT Analytics**: Servicio para el cálculo de grandes volúmenes de datos de IoT.
  - **AWS IoT SiteWise**: Servicio de adquisición y análisis de datos industriales.
  - **AWS IoT Events**: Servicio para facilitar las respuestas a eventos de grandes cantidades de aplicaciones y dispositivos IoT.
  - **AWS IoT Things Graph**: Servicio que facilita la interconexión entre diferentes dispositivos y servicios para crear aplicaciones online.

Como se puede ver en la web, es interesante destacar que grandes empresas como VolksWagen, LG, iRobot o Bayer, hacen uso de la plataforma AWS IoT para mejorar las capacidades de sus productos o incluso su productividad. En los siguientes subapartados se profundiza más en los dos servicios en los que se basa principalmente este TFG.

#### 2.3.3.1 AWS IoT Core

Como su propio nombre indica, este servicio es el núcleo de AWS IoT. Es el encargado de conectar los dispositivos a la nube de Amazon. Tal y como lo presentan en su portal [9], la principal ventaja de este servicio es no tener que administrar servidores (pudiéndo catalogarse como PaaS) además de realizar de forma casi automática la escalabilidad, admitiendo millones de dispositivos y de mensajes siendo capaz de procesarlos y reenviarlos al servicio o nodo que corresponda. Además de permitir la conexión de dispositivos, sigue sin pasar por alto la seguridad en las transmisiones permitiendo también elegir el protocolo a emplear en las mismas. Por otro lado, ofrece una gran facilidad para poder conectar con cualquier otro servicio de AWS como, por ejemplo, Lambda, Bases de Datos, S3, etc.

Dentro de las capacidades de comunicación que ofrece, esán las M2M (*Machine to Machine*) enfocadas quizá a un entorno más industrial y productivo, de forma que se automaticen las decisiones a tomar en el proceso monitorizado.

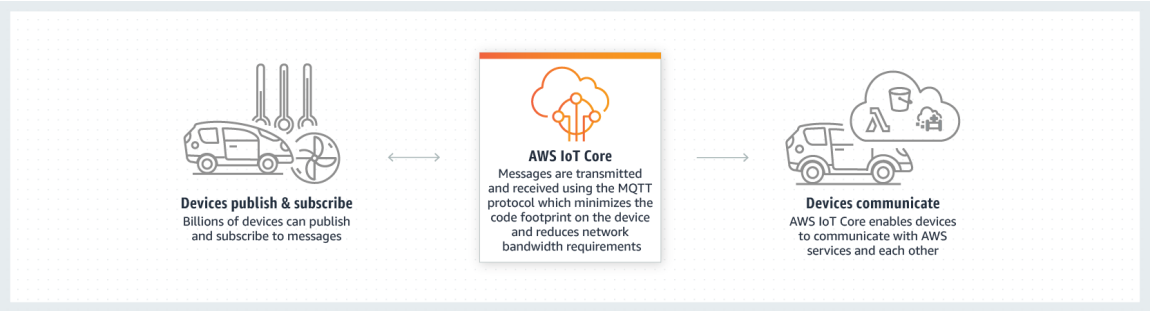


Figura 2.4: Diagrama de bloques de AWS IoT Core para M2M [41]

También ofrece la integración con Alexa Voice Service (AVS) con un dispositivo virtual en la nube. En este caso el protocolo empleado no se puede seleccionar, tendrá que ser MQTT pero da la flexibilidad de poder controlar los dispositivos con comandos de voz.

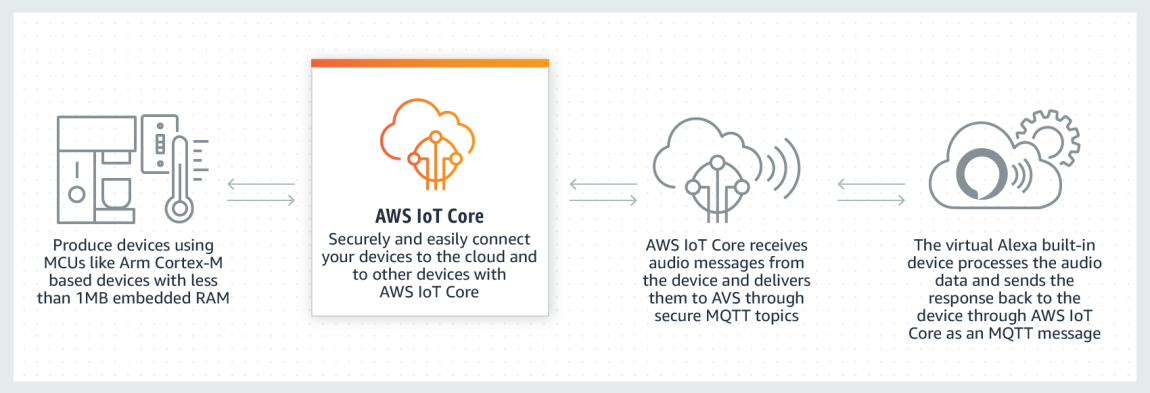


Figura 2.5: Diagrama de bloques de AWS IoT Core con Alexa [42]

Por si fuera poco, AWS IoT Core, también tiene la opción de crear una red LoRaWAN y poder conectar sus *gateways* a la nube de forma sencilla, pudiendo así, aprovecharse de las capacidades de bajo consumo y alto alcance de este protocolo de comunicaciones sin tener que administrar un LoRaWAN Network Server (LNS).

Profundizando en el funcionamiento de este servicio, la idea es sencilla: unos pequeños dispositivos electrónicos han de conectarse a un servidor "en la nube" para transmitir la información que captan (por ejemplo, temperatura y humedad relativa del aire). En principio ninguno de estos dispositivos tendría que procesar los datos, solo captarlos, transmitirlos y, como mucho obedecer órdenes emitidas por otros nodos con mayor capacidad de análisis y cálculo. Desde un punto de vista más tradicional, no se trataría ni de la topología *master-slave* ni de la *multi-master*, es más bien una topología en estrella en la que habrá nodos que transmitan lo sensado, nodos que ejecutarán órdenes y otros nodos que serán los encargados de tomar decisiones, almacenar datos... pero todos conectados al mismo punto que será el encargado de reenviar los mensajes según una filosofía de suscripción-publicación.

En AWS IoT, ese servidor en centro de la estrella que se encarga de recibir los datos de los dispositivos y reenviarlos es el nodo que gestiona todos los mensajes (en el protocolo MQTT se le llama ***broker***). Dicho nodo también puede reenviar los mensajes/datos sensados al resto de servicios de AWS mediante su motor de reglas.

AWS IoT tiene, en apariencia, un cometido bastante sencillo, pero lo cierto es que su entorno es un tanto complejo de entender en su inicio. A priori no es nada intuitivo cómo puede ser tan complicado configurar todo el sistema para que "simplemente" reciba datos. Con paciencia, una vez se realizan todos

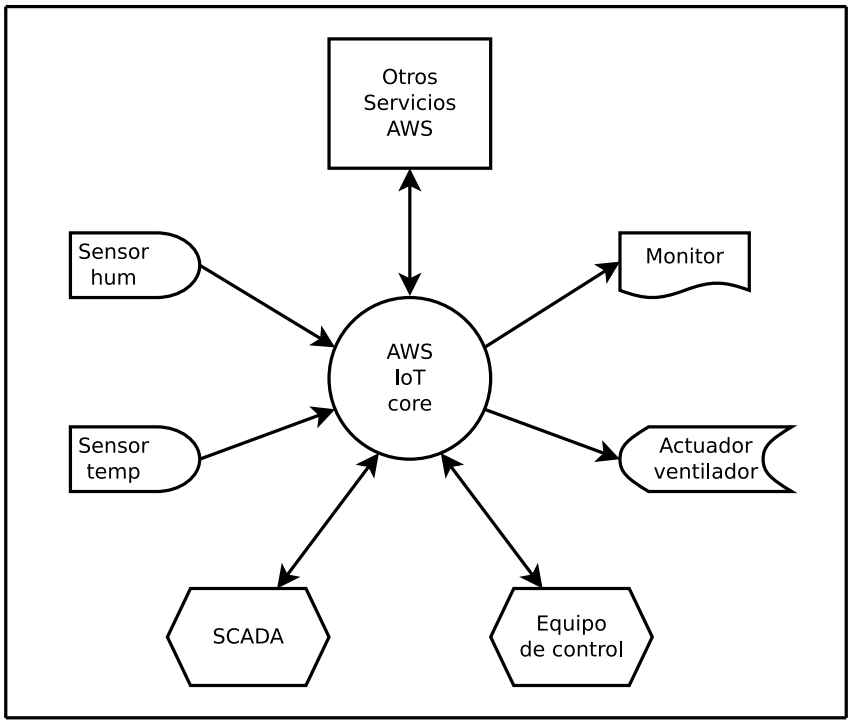


Figura 2.6: Topología AWS IoT

los pasos y se entiende el por qué de cada uno de ellos, se puede apreciar con algo más de claridad el por qué de esas, en apariencia, ofuscadas configuraciones.

Una de las primeras tareas a realizar es la creación del objeto o *thing*. Implica generar certificados, credenciales y contraseñas para poder realizar una comunicación de un objeto con AWS IoT. En [10] se describen los pasos a seguir para realizar la creación. Esas credenciales, certificados y contraseñas pertenecen a la capa de seguridad que añade AWS IoT a las comunicaciones para que puedan ser confiables. Han de guardarse en ficheros de texto a buen recaudo, pues serán necesarias más adelante.

Muchos dispositivos (cada vez menos) son de baja potencia de cálculo como para poder manejar encriptación, por eso AWS IoT exige de unos mínimos a dichos dispositivos para que las comunicaciones puedan ser seguras y fiables y no puedan ser facilmente interceptadas. Una lista de los aprobados puede verse en [5]. Obviamente, otros muchos dispositivos sin este "visto bueno de AWS" también podrán cominucarse, pero puede que no haya tantas facilidades y documentación. Aunque esta última parte parece exagerada desde el punto de vista de un sencillo TFG como este, hay que pensar que este sistema se usa a gran escala con equipos muy costosos que no se pueden permitir que puedan ser fácilmente hackeados. Un buen ejemplo de esto último es el de las minas de Riotinto, que controlan la posición de los *dumpers* de la mina y hacen que elijan la ruta más óptima (mediante análisis y algoritmos) haciendo que se ahorre mucho en gastos de combustible y mantenimiento.

Por otro lado, hay que crear una/s política/s para dar permiso o no a lo que se asocie a ellas. En [11] se explica paso a paso cómo crearlas. Es un concepto un tanto abstracto que termina de cobrar significado cuando se ve que la finalidad de la política es (entre otras) que un objeto se asocie a ella para que pueda tener permisos. Dichos permisos serán para poder enviar información a AWS IoT (más tarde se verá que esta acción se llama 'publicar'), poder recibir esa información (concepto de 'suscripción') listar las distintas informaciones disponibles, etc.

Una vez asociados los objetos con las políticas, hay que comenzar a recibir datos en AWS IoT. La plataforma cuenta con herramientas de monitorización para ver que están llegando los datos enviados

por los objetos y lo que se está respondiendo. Desde la "Consola de AWS IoT" en la sección "Prueba" (o "*Test*" para la versión en inglés) se accede a la herrameinta "Cliente de prueba" donde se puede monitorizar.

Hay muchos más aspectos que se podrían comentar como que, por ejemplo, AWS IoT tiene integrado un *gateway* que se encarga de la comunicación bidireccional entre los dispositivos y el *core*, pero en realidad este es prácticamente transparente para la realización de este TFG y para un uso normal de este entorno, por lo que profundizar en este aspecto es añadir más carga innecesaria a este texto.

Las comunicaciones entre AWS IoT Core y los dispositivos pueden realizarse (Hasta el momento de la realización de esta memoria) mediante tres protocolos:

- HTTP
- WebSockets
- MQTT

Como se indicó con anterioridad, las comunicaciones con AWS IoT simepre irán protegidas con TLS. Quizá este sea uno de los aspectos más interesantes de AWS IoT: en ningún momento, independientemente del tamaño del sistema implementado, las comunicaciones se harán sin proteger.

En cuanto a los protocolos que se pueden emplear, decir que tanto HTTP como WebSockets son perfectamente válidos y útiles, pero en este TFG, las comunicaciones emplearán el protocolo MQTT. La elección del mismo se debe a la facilidad de empleo del mismo pues es *human readable*, la cantidad de documentación al respecto y, sobre todo, que se trata de un estándar *de facto* para todo lo relacionado con IoT.

#### El servicio *shadow* de AWS IoT

Este (sub)servicio de AWS IoT es tan interesante como importante[4].

La sombra o *shadow* de un dispositivo AWS IoT es un fichero `.json` (explicado en la sección 2.5) que hace de intermediario para que objetos y aplicaciones puedan acceder al estado del dispositivo independientemente de si están o no conectados a internet. Se trata, pues, de una "frontera" que abstrae las comunicaciones entre dispositivos a los objetos lógicos con que trata AWS IoT. La utilidad de que no importe si hay o no conexión a internet por parte del dispositivo es doble:

- Se podrá seguir interactuando con dispositivos que tengan un ancho de banda estrecho o con una alimentación eléctrica un tanto deficiente que haga que sólo se pueda conectar de forma intermitente. En el momento que se conecte, se actualizará toda la información necesaria.
- Se puede simular la existencia de un objeto con su sombra sin la necesidad de realmente exista. Al poder interactuar con ella, es posible desarrollar sin que haya un hardware conectado.

Este servicio se comunica a través del protocolo MQTT (se verá en la sección 2.4) y mantiene unos cuantos *topics*[5] para facilitar la comunicación entre este dispositivo y otros o entre aplicaciones y dispositivo de forma sencilla.

[4] Para más información: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/iot-device-shadows.html – Última visita: septiembre de 2020.

[5] Ver todos los *topics* en https://docs.aws.amazon.com/es_es/iot/latest/developerguide/device-shadow-data-flow.html

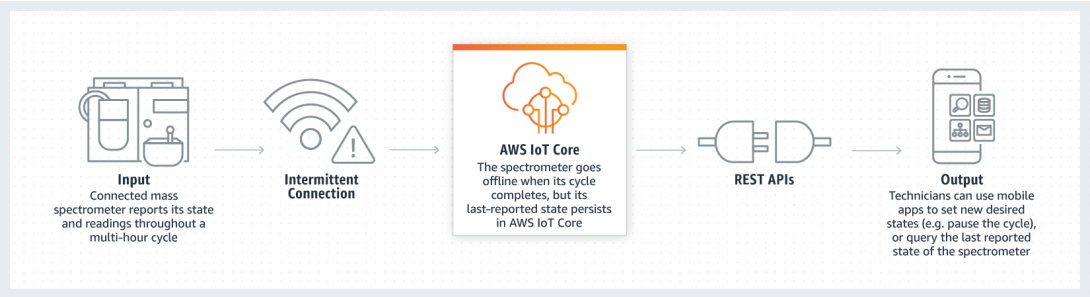


Figura 2.7: Diagrama bloques servicio *shadow* [43]

#### 2.3.3.2 Servicio Amazon FreeRTOS

Un Sistema Operativo en Tiempo Real (RTOS) se puede entender como un sistema operativo que no tiene que ser muy eficiente en términos de potencia de cálculo (no tiene que ser rápido) pero si tiene que poder atender a diferentes tareas e interrupciones de forma que, cumpliendo una temporización estricta, obtenga resultados de forma predecible y en tiempo finito. En un sistema en tiempo real, se pueden ejecutar varias tareas "al mismo tiempo"[6] (como en un Sistema Operativo de Tiempo Compartido) pero, sobre todo, tendrá que interactuar con el mundo real a través de eventos e interrupciones externas. Al tratarse de un sistema determinista, cada tarea y/o interrupción sólo dispondrá de un tiempo determinado para poder realizar los cálculos y ejecutar los algoritmos antes de ceder el testigo a otra tarea. Los sistemas operativos no deterministas, no aseguran esta temporización. Sin profundizar mucho más, cabe destacar que existen dos tipos de RTOS:

- Guiado por eventos: el cambio de tarea se produce cuando un evento necesita ser atendido.
- Compartición de tiempo: el cambio entre tareas se produce cuando lo determina el reloj o cuando se produce un evento al que atender, p.ej: una interrupción.

Otra característica a tener en cuenta en los RTOS es la "rigidez" de la temporización, pueden ser:

- Flexibles: Si, pasado el tiempo determinado para la tarea, no se ha obtenido un resultado, no es una circunstancia crítica, no hay mayor problema. Un ejemplo sencillo: un control de horneado: si tarda 2 ms más en apagar el horno, el asado no sufrirá gran pérdida de sabor.
- Estrictos: Deben garantizar un resultado válido en un tiempo determinado pues, de no obtener un resultado, es un problema crítico. Un ejemplo clásico: un sistema de guía de un misil anti-misil.

Como apunte final, para poder cumplir los requerimientos temporales hay que evitar emplear algoritmos y funciones que tengan una duración poco predecible. Por ejemplo, en C/C++ todo lo que se refiere a reserva dinámica de memoria [12].

#### FreeRTOS™

Si bien, en la sección anterior, se describe de forma conceptual lo que es un RTOS, no queda del todo definido sobre qué soporte físico va a funcionar. Puede ser un PC, un *mainframe* o un servidor sencillo. FreeRTOS hace que se puedan cumplir todos esos requisitos pero corriendo en un microcontrolador, no en un hardware complejo. Obviamente un microcontrolador de 8 bits es complicado que sea capaz de hacer funcionar de forma eficiente un RTOS, aunque hay proyectos que los hacen funcionar.

[6] Conviene recordar que en sistemas con un único procesador, en realidad no se pueden ejecutar varias tareas a la vez, es el cambio de atención del procesador a una tarea u otra lo que da sensación de multitarea.


Figura 2.8: Logo FreeRTOS™[44]

Desde [13] se puede descargar el kernel, este es distribuido de forma gratuita con la licencia de código abierto MIT. En dicha web también se puede acceder a mucha información, ejemplos, librerías, entrar en foros... Una sección interesante es la que lista los dispositivos que soportan FreeRTOS.

#### Amazon FreeRTOS

Amazon FreeRTOS [14] es un sistema operativo en tiempo real para microcontroladores **basado** en FreeRTOS. Sobre esta base, Amazon ha añadido funcionalidades extra para facilitar las comunicaciones entre el dispositivo y AWS IoT y, así, ha convertido FreeRTOS en el servicio Amazon FreeRTOS. En otras palabras, **el servicio Amazon FreeRTOS es FreeRTOS con unas capas extra de comunicaciones seguras**.

De igual forma que el sistema del que deriva, Amazon FreeRTOS está orientado a microcontroladores y dispositivos poco potentes y/o de bajo consumo. Existe una lista [15] con los dispositivos probados y, de alguna forma, certificados para poder correr el SO, pero AWS también ofrece la posibilidad de realizar unas pruebas para validar un chip que no esté en la lista anterior y asegurarse de que puede funcionar con su FreeRTOS [16].

### 2.3.4 Servicio AWS Lambda

El servicio AWS Lambda [17], también conocidas como funciones Lambda, no forma parte de la plataforma AWS IoT. Es un servicio *serverless* que permite ejecutar código casi para cualquier aplicación sin tener que hacer tareas de administración.

Figura 2.9: Logo AWS Lambda [45]

Para que se ejecute el código han de cumplirse unas condiciones llamadas “desencadenadores” que pueden provenir desde casi todos los servicios de AWS. Un ejemplo sencillo de desencadenador es que se reciba un mensaje en AWS IoT. De esta forma al recibirse el mensaje, AWS ejecutará el código que puede estar escrito en Python, Node.js, Go, Java...

La gran ventaja de este servicio Lambda es que el código no está escrito en ningún sitio concreto, es decir, es un servicio independiente del resto de servicios AWS y siempre podrá estar activo sin tener que preocuparse por administrar la máquina ni la ubicación donde esté.

### 2.3.5 No-servicio de gráficos

A pesar de la gran cantidad de servicios que hay en AWS, no hay –a día de escribir esta memoria– un servicio que muestre los datos en forma de gráfico $xy$, algo sencillo que, a priori, no debería ser muy complicado, no está en su oferta. Si ofrece gráficos analíticos para estudios de inteligencia comercial como *Amazon QuickSight* pero no está orientado a mostrar datos en ejes, está más pensado para análisis empresarial donde otro tipo de gráficos son un elemento de apoyo a las estadísticas y conclusiones realizadas. Además no está en la *free tier*. Debido a ello, para mostrar de forma gráfica las mediciones de los sensores, se emplearán librerías ofrecidas por AWS para *Python* y poder mostrarlas de forma local accediendo a los datos de la nube, demostrando así que, a pesar de lo cerrado que puede parecer la plataforma de Amazon Web Services, da muchas opciones de accesibilidad haciendo que las posibilidades se incrementen de forma exponencial.

### 2.3.6 Bases de datos en AWS

El escenario es el siguiente: el *edge device* ya ha adquirido las mediciones, las transmite al *core* de AWS IoT y es el momento de almacenarlas para que no se pierdan, pues una vez transmitidas, no se guardan sin más. Hay que configurar AWS para que lo haga. Es el turno de hablar de los servicios de bases de datos. AWS tiene muchos servicios de bases de datos. En [18] se muestran más de diez servicios de bases de datos en el momento de escribir estas líneas con muy diversas capacidades y características. Tal y como reza en la web de presentación de las bases de datos de Amazon[7]: "*Elegir la mejor base de datos para solucionar un problema o un grupo de problemas específicos le permite olvidarse de las bases de datos monolíticas, genéricas y restrictivas, y centrarse en el desarrollo de aplicaciones que satisfagan las necesidades de su negocio*".

Figura 2.10: Logo AWS DynamoDB [46]

En este punto, la dificultad está en elegir bien qué base de datos se adapta mejor a las necesidades de almacenar datos provinientes de IoT. AWS ofrece opciones de bases de datos relacionales de gran potencia y escalabilidad pero, la que ofrece mayores facilidades –en la capa *free tier*– a la hora de almacenar tanto mensajes MQTT como registros, es *DynamoDB* [19]. Se trata de una base de datos propia de AWS, es NO relacional, NoSQL, totalmente administrada, flexible, escalable, potente y que soporta una carga de consultas y datos muy alta. También es destacable la sencillez de crear una tabla de datos para comenzar a introducir registros en ella así como de características más complejas como el cifrado para información confidencial.

[7] En la dirección: https://aws.amazon.com/es/products/databases/ – Último acceso: septiembre 2020.

## 2.4 El protocolo MQTT

El nombre de este protocolo proviene de *Message Queuing Telemetry Transport* [20] y es un estándar ISO (ISO/IEC 20922:2016)[21].


Figura 2.11: Logo MQTT [47]

Se trata de un protocolo ligero para intercambiar información entre máquinas (*Machine to Machine* o M2M), basado en publicación-suscripción (también funciona sobre TCP/IP). El protocolo fue creado por Andy Stanford-Clark y Arlen Nipper para la industria petrolera, para poder comunicar sensores instalados a lo largo de los oleoductos que atraviesan desiertos. Buscaban bajo coste y mantenimiento tanto para los sensores como para las comunicaciones. El protocolo pertenecía a IBM pero fue liberado en 2010. La información transmitida con este protocolo suele ser reducida[8] de tal forma que no necesite mucho código para interpretarla ni tampoco un gran ancho de banda que suele traducirse en un menor consumo de energía tal y como se buscaba cuando se creó. Con posterioridad, el auge de los dispositivos IoT que necesitan estar conectados pero no suelen disponer de una alimentación de gran capacidad, hizo que este protocolo fuera el más utilizado para las comunicaciones de el Internet de las cosas.

El protocolo consiste en la comunicación de los distintos dispositivos o clientes con un servidor o *broker*. Que se trate de un protocolo de publicación-suscripción implica que un cliente que se conecte al *broker* puede publicar una información en un tema o *topic*, suscribirse para recibir información publicada por otro cliente en un tema o también publicar y suscribirse al mismo tema o diferentes. Un tema o *topic* es como un apartado de correos: alguien deja una información en el buzón y esta información se distribuye a quien esté suscrito a dicho apartado. Los temas están organizados de forma jerárquica, de tal forma que si un dispositivo está suscrito a un tema del que dependen más, recibirá información de todos y cada uno de dichos temas. Si el *broker* recibe información en un *topic* que no tiene suscriptores, se descartará salvo que quien publica indique que tenga que ser retenido, de esta forma, futuros suscriptores no perderán información.

Se pueden configurar muchos aspectos y parámetros de este protocolo como, por ejemplo, mensajes predeterminados que enviar a los clientes suscritos a un tema si se detecta que el cliente que publica se ha desconectado. Aunque los clientes nunca interactúan entre sí, pues sólo lo harán con un *broker*, dentro de un sistema puede haber más de un *broker* que intercambiarán la información para poder entregarla a los diferentes suscriptores de un tema.

En lo referente a calidad/integridad de los mensajes, el protocolo define un mecanismo de "calidad del servicio" llamado QoS (*Quality of Service*) en tres niveles para la actuación frente a fallos de comunicación:

- QoS 0 unacknowledged (at most one): el mensaje se enviará una vez pero, al no esperarse confirmación, si algo falla es posible que se pierda el mensaje y no sea entregado.
- QoS 1 acknowledged (at least one): se enviará el mensaje las veces que sean necesarias hasta que se reciba confirmación. Si lo que falla es, por ejemplo, la confirmación, puede que le lleguen al destinatario varias veces el mismo mensaje.

[8] El mensaje más pequeño contendrá tan solo dos bytes aunque puede llegar a los 256 MB.

- QoS 2 assured (exactly one): se garantiza que el mensaje llega al destinatario y una única vez.

A mayor QoS habrá más seguridad de que sea entregado/recibido el mensaje, pero también ocupará más tiempo el canal de comunicaciones y requerirá de mayor procesamiento por parte de los dispositivos implicados.

En cuanto a la seguridad, MQTT envía la autenticación (usuario y contraseña) en texto plano y no incluye mucha más seguridad en cuanto a autenticación. Es por eso que tendrá que ser la capa TCP/IP la que añada esa seguridad extra si se desea. Lo más habitual es emplear SSL/TLS.

En el momento de escribir este TFG, coexisten dos versiones del protocolo MQTT, la 3.1.1 y la 5.0, ambas ratificadas por ISO. El protocolo MQTT tiene como puerto estándar TCP/IP el 1883 y el 8883 para usarlo sobre SSL. A pesar de que está estandarizado para ser usado sobre TCP/IP, también existen variantes especialmente pensadas para usarse con UDP o con Bluetooth.

## 2.5 Los ficheros `.json`

El nombre de este tipo/formato de archivo viene de **J**ava**S**cript **O**bject **N**otation. Aunque originalmente se pensó para JavaScript, es independiente y se ha convertido en un formato estándar abierto de texto plano para el intercambio estructurado de datos similar al formato XML[22] pero con la –no única– ventaja de que es *human readable* y también es más fácil implementar un analizador o *parser* para evaluar el contenido de dicho fichero.

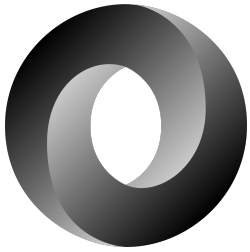

Figura 2.12: Logo JSON [48]

Un fichero `.json` puede contener los siguientes tipos de datos básicos en su interior[23]:

- Números: Pueden ser tanto enteros (positivos y negativos) como decimales, donde el separador entre la parte entera y la mantisa, será un punto.
- Cadenas: Representan secuencias de caracteres. Se ponen entre doble comilla (p.ej. “Hola”). Se permiten caracteres de escape y también cadenas vacías (p.ej. “”).
- Booleanos: los típicos valores booleanos: *true* y *false*
- Valor ‘*null*’: es un valor especial parecido a los booleanos, pero este representa el valor nulo.

A su vez, puede contener estructuras más complejas –incluso anidadas– formadas con los tipos básicos, como son:

- Array: Encerrado entre corchetes, contiene una lista ordenada. Los valores pueden ser de cualquier tipo de los anteriores y se separan por comas. La lista puede no tener valores.

- Objetos: Son colecciones no ordenadas de pares de valores con la forma <nombre>:<valor> donde cada par estará separado por comas y puestas entre llaves. El <nombre> tiene que ser una cadena. El <valor> puede ser de cualquier tipo.

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
 }
}
```

Listado de código 2.1: Ejemplo fichero `.json`

Ni qué decir tiene que, a pesar de que se puedan formar estructuras de datos bastante complejas, el fichero en sí será bastante ligero, ideal para poder ser transmitido por dispositivos con un ancho de banda escaso y una potencia –tanto de cálculo como de alimentación– limitada. Es por esto que la información a transmitir por los dispositivos IoT, bien sean parámetros capturados/medidos por los sensores y/o las acciones tomadas, se estructurará en un fichero de este tipo y, una vez generado, será finalmente enviado a AWS IoT mediante el protocolo MQTT, sin olvidar que irá encriptado con TLS. Del mismo modo, la información retransmitida a los suscriptores del tema también será un fichero `.json`.

## 2.6 Lenguaje *Python*

El lenguaje Python [24] necesita poca presentación. Comentar de forma escueta que es un lenguaje de programación interpretado pero que también acepta *scripts*. Soporta programación orientada a objetos, programación imperativa, es código abierto y multiplataforma. Esto último hace que los scripts que se generen en Linux se puedan ejecutar en Windows sin apenas tocar nada del código. Es muy fácil comenzar con él, pues con unos sencillos tutoriales ya se podrán programar scrips sencillos e ir aumentando la complejidad muy rápidamente. Por todo ello se ha ido ganando los puestos de lenguaje más empleado y querido para programar en los últimos años. Un detalle más: las funciones del sevicio *AWS Lambda* se pueden escribir en Python.

Figura 2.13: Logo Python [49]

AWS ofrece muchas librerías para poder acceder a distintos servicios mediante scrips/programas escritos en Python. Las librerías que se emplearán en este TFG son las de acceso a IoT, para poder

suscribirse y publicar topics y, como se verá más adelante, poder representarlos de forma gráfica *xy*. Un aspecto común que tendrá todo el código –no solo el de Python– que desée acceder a AWS son los certificados, hay que incluir los certificados generados en AWS (tanto CA como claves de objetos) dentro del programa para poder acceder de forma segura a la nube.

## 2.7 Entorno de desarrollo (IDE)

Se opta *por System Workbench for STM32* [25]. Al tratarse de una herramienta *open source* y estar basado en Eclipse[9] es compatible con el sistema operativo Linux con el que se está desarrollando todo el trabajo.


(a) Logo Eclipse [50] (b) Logo OpenSTM32 [51] (c) Logo AC6 Tools [52]

Figura 2.14: Logos de las partes que intervienen en el IDE

Este IDE está desarrollado por la empresa AC6 y da soporte a través de su web y sus foros a todos los usuarios que lo requieran. Solo es necesario registrarse en su web (www.openstm32.org) para poder descargar el entorno y acceder a todos sus contenidos. Por otro lado, también es una de las opciones que ofrece el servicio Amazon FreeRTOS para poder descargar los ejemplos y poder desarrollar las aplicaciones deseadas.

## 2.8 Hardware

Se seleccionan dos tarjetas de desarrollo para la realización de pruebas de comunicación sencillas. Una de ellas, además, será la empleada en el posterior desarrollo completo del TFG.

### 2.8.1 ESP32 Devkit C

Se elige una tarjeta muy extendida, asequible y con mucha documentación en Internet. El modelo concreto es la versión "ESP WROOM 32" que se puede ver en la Figura **??**. Lleva integrados módulos de WiFi, Bluetooth y BLE MCU. Sus principales características son:

- 18 canales de conversión analógico-digital (ADC)
- 10 GPIO de detección capacitiva
- 3 interfaces UART
- 3 interfaces SPI

[9] Web del IDE Eclipse: https://www.eclipse.org/ide/

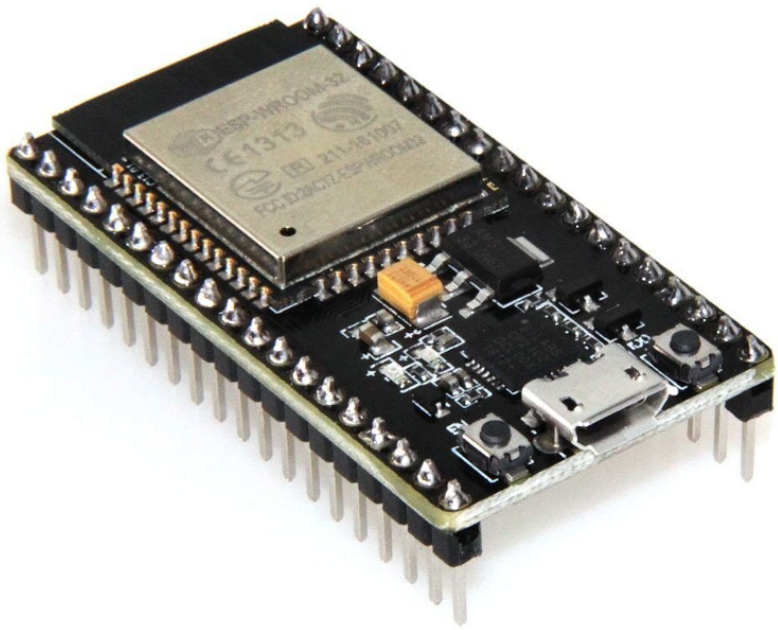

Figura 2.15: ESP32 Devkit C [53]

- 2 interfaces I2C
- 16 canales de salida PWM
- 2 Convertidores de digital a analógico (DAC)
- 2 interfaces I2S

Esta tarjeta de desarrollo está especialmente diseñada para adaptarse a una amplia variedad de aplicaciones como redes de sensores IoT hasta diseños más complejos como la codificación de voz. La versatilidad de su huella de paso 100 mills, el hecho de que ya tiene integrados periféricos *wireless*, su plena integración en el IDE de Arduino (entre otros) junto a la gran documentación, foros y ejemplos que hay en Internet, le otorgan una gran flexibilidad y velocidad en el desarrollo de proyectos.

### 2.8.2 STM32L4 Discovery kit for IoT node

Se podría catalogar como una tarjeta de gama media-alta debido a que, en el momento de salir al mercado, el microcontrolador que monta era uno de los que mejores características ofrecía en cuanto a bajo consumo *vs* rendimiento. Además de ello cuenta con gran cantidad de sensores y periféricos montados alrededor del mismo que ofrecen multitud de posibilidades. Sus principales características (extraídas de [26]):

- Microcontrolador de ultra bajo consumo STM32L4 basado en Arm Cortex®-M4 core con 1 Mbyte de memoria Flash y 128 Kbytes de SRAM, en formato LQFP100
- Memoria flash de 64-Mbit Quad-SPI (Macronix)
- Módulo Bluetooth V4.1 (SPBTLE-RF)
- Módulo RF de bajo consumo Sub-GHz (868 MHz or 915 MHz) (SPSGRF-868 o SPSGRF-915)
- Módulo WiFi 802.11 b/g/n de Inventek Systems (ISM43362-M3G-L44)

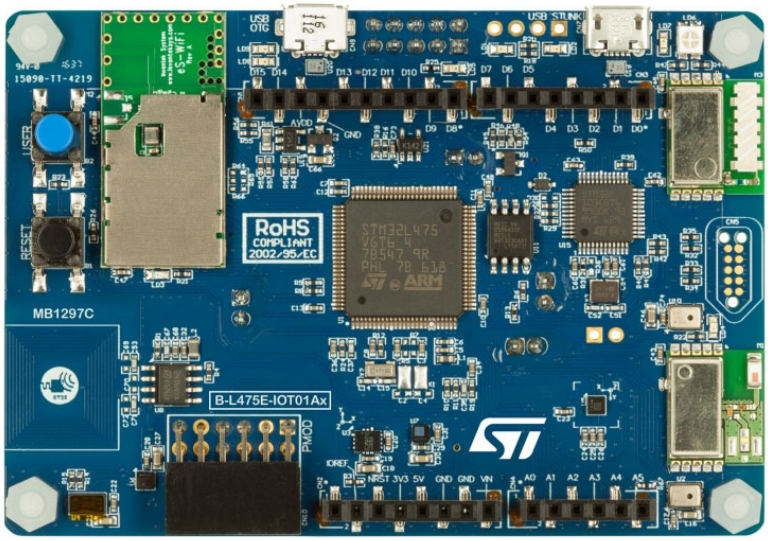


Figura 2.16: STM32L4 Discovery Kit [54]

- *Tag* NFC dinámica basado en M24SR con su propia antena impresa en pista
- Dos micrófonos digitales omnidireccionales (MP34DT01)
- Sensor capacitivo digital para humedad relativa y temperatura (HTS221)
- Magnetómetro de 3 ejes de alto rendimiento (LIS3MDL)
- Acelerómetro y giróscopo 3D (LSM6DSL)
- Barómetro digital absoluto de 260-1260 hPa (LPS22HB)
- Time-of-Flight (ToF) y sensor de detección de gestos (VL53L0X)
- Dos pulsadores (usuario y reset)
- Conector Micro-AB USB OTG FS
- Zócalos de expansión para:
  - Arduino Uno V3
  - PMOD
- Opciones de alimentación flexible: ST LINK USB VBUS o fuentes externas
- On-board ST-LINK/V2-1 programador/depurador con capacidad de re-enumeración USB: almacenamiento masivo (mass storage), puerto virtual COM y puerto de depuración

Es destacable también, aunque no aparezca en la presentación de sus características, que el microcontrolador posée un RTC interno y que la tarjeta también dispone del oscilador de 32 KHz para su correcto funcionamiento.

En cuanto a la parte software:

- Software completo y gratuíto de librerías HAL incluyendo variedad de ejemplos como parte de "STM32Cube MCU Package"
- Amplio soporte de gran variedad de IDE's incluyendo: IAR, Keil, GCC-based IDEs, Arm Mbed Enabled (aunque no lo dice expresamente, también es compatible con el IDE *System Workbench for STM32*)
- Arm Mbed online (ver http://mbed.org)

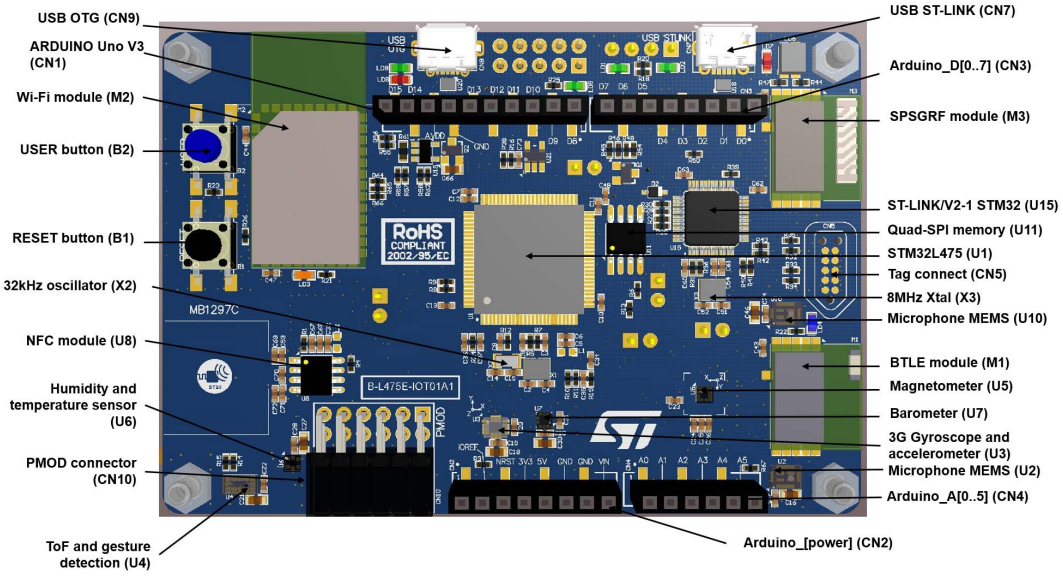


Figura 2.17: Dibujo elementos STM32 Discovery IoT [55]

La forma de acceder a los periféricos y sensores, como se puede ver en la Figura **??**, es variada: un puerto QSPI (para una memoria Flash QSPI), un $I^2C$ para acceder a la mayoría de los sensores, pines GPIO, SPI...

### Sensor capacitivo digital para humedad relativa y temperatura (HTS221)

Será este sensor [56] integrado en la de la tarjeta tarjeta *Discovery kit* el usado para realizar las mediciones de las magnitudes temperatura y humedad.

Sus principales características son:

- Rango de medición de humedad relativa de 0 a 100 %
- Alimentación: 1.7 a 3.6 V
- Bajo consumo: 2 $\mu$A @ 1 Hz ODR
- ODR seleccionable desde 1 Hz a 12.5 Hz
- Alta sensibilidad rH: 0.004 % rH/LSB
- Precisión en la humedad: ± 3.5 % rH, 20 a +80 % rH
- Precisión en la temperatura: ± 0.5 °C,15 a +40 °C
- ADC 16-bit Integrado

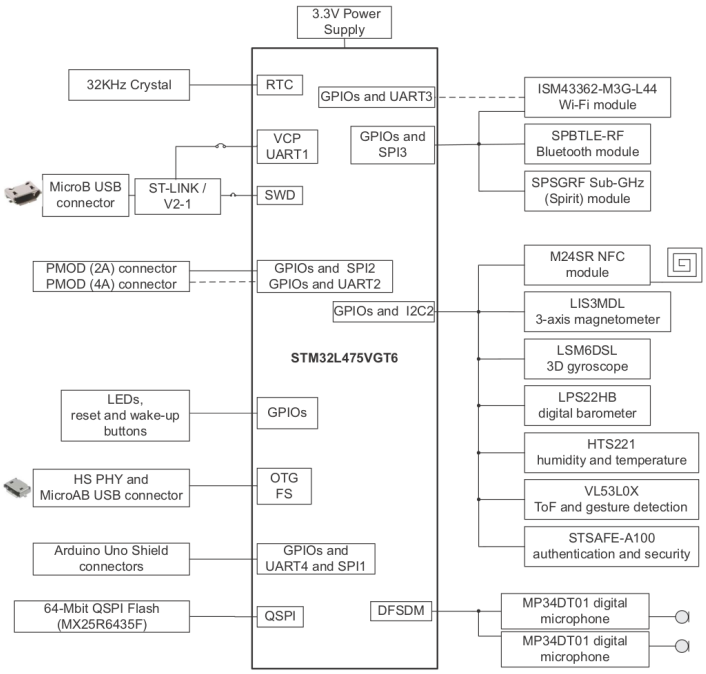


Figura 2.18: Diagrama de bloques STM32 Discovery IoT [55]


Figura 2.19: Sensor de temperatura y humedad HTS221 [56]

- 16-bit humidity and temperature output data
- Interfaces $I^2C$ y SPI
- Calibrado de fábrica
- Empaquetado pequeño de 2 x 2 x 0.9 mm
- Cumple ECOPACK

Físicamente se trata de un único circuito integrado, pero desde el punto de vista lógico se trata cmo si fueran dos. Para poder acceder a sus funciones, habrá que incluir como cabeceras los *headers* del Listado 2.2. Para realizar las medidas, se profundizará más en la Sección 3.6.6.

```
#include "cmsis_os.h"
#include "stm32l475e_iot01.h"
#include "stm32l475e_iot01_tsensor.h"
#include "stm32l475e_iot01_hsensor.h"
```

Listado de código 2.2: *Headers* para los sensores de temperatura y humedad

Donde:

- `cmsis_os.h`: Incluye las descripciones hardware de bajo nivel.
- `stm32l475e_iot01.h`: Incluye las macros y etiquetas que usan los sensores.
- `stm32l475e_iot01_tsensor.h`: Incluye las funciones para la medición de temperatura.
- `stm32l475e_iot01_hsensor.h`: Incluye las funciones para la medición de humedad.

# Capítulo 3

# Implementación práctica

## 3.1 Introducción

Tras la explicación teórica de cada una de las partes de este TFG y antes de exponer el desarrollo propiamente dicho en los dos últimos apartados de este capítulo, se describirá someramente cómo funcionan dichos elementos. Se hará mediante sencillos ejemplos prácticos.

## 3.2 Protocolos

### 3.2.1 El protocolo MQTT con *mosquitto*

Como primer elemento tenemos el protocolo especialmente pensado para IoT. Se mostrará, de forma rápida, cómo funciona el protocolo MQTT realizando un ensayo: En una Raspberry Pi conectada a la misma red que el PC que se usará para la prueba, se instala el programa *Eclipse Mosquitto*[28] en su versión server. Este programa es un *broker* MQTT de código libre y potente con el que poder montar una red de dispositivos que se interconectan con dicho protocolo. Del mismo modo que se instala el servidor en la Raspberry Pi, hay que instalar el cliente del mismo en el PC con el que trabajaremos.

Con el cliente instalado en el PC y el servidor instalado y corriendo en la Raspberry Pi en la dirección IP 192.168.1.200, abriremos dos terminales en el PC: uno simulará el sensor que informa sobre el estado de la puerta y otro el que estará pendiente de dicho estado para poder realizar otras tareas como, por ejemplo, activar un motor para abrir o cerrar según convenga.

Como se vio en el marco teórico, un broker MQTT funciona mediante publicaciones y subscripciones. Antes de publicar nada, vamos a activar la escucha (≡ suscribir) escribiendo en el terminal que simulará estar pendiente del estado de la puerta:

```
$ mosquitto_sub -h 192.168.1.200 -t casa/cocina
```

Listado de código 3.1: Ejemplo mosquitto sub

... y, de esta forma, permanecerá a la escucha hasta que lleguen los mensajes.

Por otro lado, en el otro terminal simularemos el envío –por parte de un sensor– el estado de la puerta, realizará una publicación en el topic o tema donde está escuchando el otro 'ispositivo". En el terminal escribiremos:

```
$ mosquitto_pub -h 192.168.1.200 -m "Puerta abierta" -t casa/cocina
```

Listado de código 3.2: Ejemplo mosquitto pub

Acto seguido, veremos como en el primer terminal, el suscriptor, aparece "`Puerta abierta`" en texto plano, sin más... bueno, habiendo pasado el mensaje por el broker. Como vemos es una forma de enviar mensajes relativamente cortos que no necesitan gran ancho de banda de forma sencilla[1]. Este funcionamiento del protocolo MQTT es la base de funcionamiento del sistema AWS IoT.

### 3.2.2 Protocolo MQTT y AWS IoT

Tras el primer contacto de la sección anterior, el siguiente paso será enviar un mensaje sencillo mediante MQTT a AWS IoT. Para ello serán necesarios varios pasos:

- Configurar el entorno AWS IoT para que acepte mensajes. Sin olvidar que hay que crear también los certificados que añadirán la capa se seguridad en la transmisión de AWS. Para ello, se creará una política que permita acceso total a la parte de AWS IoT[2] así como generar los certificados necesarios para poder acceder al servicio y que lleguen los mensajes MQTT a AWS IoT. Para este sencillo experimento no es necesario crear un objeto como tal, pues sólo con las políticas y la dirección del punto de acceso es más que suficiente.
- En el PC local, instalar y configurar un cliente que se suscriba y publique mensajes MQTT al perfil creado en AWS IoT. Para esta segunda parte, se elige el software *MQTT.fx*. Configurarlo es bastante sencillo siguiendo sus tutoriales.

Una vez se han realizado los pasos anteriores, es momento de enviar mensajes MQTT desde el cliente a la plataforma y comprobar que todo funciona. Para ello, en el cliente hay que estar conectado, es decir que se ponga en verde (lo cierto es que con esto ya hay mucho trabajo hecho y se ha conectado con AWS IoT correctamente). A continuación, se pincha sobre la opción *Publish* y se escribe el nombre del tema en el que se quiere publicar. Opcionalmente se puede elegir el nivel de QoS que se desée. Después se escribe un fichero `.json` en el área destinado a ello y se pincha en "publicar". ¿Cómo saber que ha llegado a su destino? Desde la consola de AWS IoT hay que pinchar en "Pruebas", escribir el mismo tema que el cliente (donde ha publicado el `.json`) y aparecerá una especie de terminal donde irán apareciendo los mensajes que se publiquen desde el cliente.

Con este ejercicio se ha conseguido:

[1] Se muestra la forma más sencilla, pero se puede ajustar con muchos parámetros como el QoS que asegura la recepción, la codificación del mensaje, enviar un fichero `.json`... pero hacen que el mensaje se haga más complejo

[2] Aunque sea matar moscas a cañonazos pues se está permitiendo todo cuando sólo se van a enviar unos poco mensajes, hace más sencillo el experimento pues no hay que profundizar en detalles más profundos.

- Comprobar que la multitud de cosas/configuraciones/generaciones que hay que hacer en AWS IoT funcionan y se han hecho correctamente. Se ha conseguido hacer funcionar la maquinaria de AWS IoT.
- Comprobar que un cliente MQTT externo, es decir software ajeno a AWS funciona y comunica.

En esta sección se ha usado un PC con un software para simular un dispositivo cualquiera (léase objeto) que ha enviado unos mensajes a la plataforma y han llegado. ¿Cuál sería el siguiente paso? Hacer que no sea un PC si no un dispositivo autónomo con mirocontrolador el que haga llegar los mensajes a AWS IoT.

## 3.3 Primera prueba con hardware: STM32L4 Discovery Kit y AWS IoT

Nada más adquirir el STM32L4 Discovery Kit, este viene con un firmware precargado de demostración para mostrar sus capacidades de sensado y de comunicación con AWS IoT. Para hacerlo funcionar correctamente[3], se conecta un cable USB-microUSB del PC a la placa de desarrollo en su puerto ST-LINK. El PC reconocerá automáticamente el dispositivo como `ttyACM0` (no hay que olvidar que este TFG se desarrolla con un sistema Linux Debian). Será necesario que el usuario tenga permisos para poder usar este dispositivo. Se escoge la herramienta/aplicación `CuteCom` para realizar esta conexión entre PC y STM32L4 Discovery Kit, pero puede usarse cualquier otra que permita comunicaciones con un puerto serie. La configuración básica del puerto `/dev/ttyACM0` será 115200-8-N-1. Por ahora la parte *hardware* está terminada.

Esta placa de desarrollo se comunica con la red local (y por tanto con Inernet) a través de un enlace WiFi, por lo que será necesario tener, al menos, un punto de acceso WiFi y los datos necesarios para poder conectarnos:

- SSID
- Tipo de encriptación
- Contraseña

No hay que perder la perspectiva que el objetivo de lo que se está haciendo en esta sección es comunicar el kit de desarrollo con AWS IoT con lo que, para conectar, será necesario:

- Creación de un objeto
- Certificados del objeto
- Certificado CA

En referencia al último punto, hasta ahora se han estado usando los certificados RSA de 2.048 bits pero la placa de desarrollo –al menos con la que se ha utilizado para este TFG– o quizá su firmware, no soporta este tipo de certificados y habrá que generar y usar el ECC de 256 bits que Amazon denomina "*Amazon Root CA 3*". No hay que olvidar activar los certificados para que AWS IoT admita la conexiones y los mensajes.

[3]Escribir enlace/s a webs donde hay tutorial/es.

También hay que asociar una política al objeto creado. Puede ser la misma que se creó en secciones anteriores u otra nueva, pero lo ideal es que también tenga todos los permisos en AWS IoT y así minimizar los posibles errores. Una vez funcione todo se podrán restringir los permisos para que sólo acceda a lo que necesite.

Una vez registrado el objeto al final de su creación hay que activar el certificado del mismo (si, otra vez) en el apartado Seguro → Certificados. Para este punto es recomendable apuntar el nombre del certificado en el momento de la creación pues no es nada significativo y así evitar activar otro certificado de otro dispositivo en caso de tener más.

Un último dato que hay que apuntar: el punto de acceso del objeto será también necesario para la configuración del STM32L4 Discovery Kit, pues se trata de la dirección a la que tratará de conectarse con los certificados... una vez haya establecido enlace WiFi.

Es el momento de introducir los certificados y demás datos en el dispositivo. Siguiendo las instrucciones del mismo, una vez conectado al terminal serie, hay que resetear el dispositivo pulsando el botón negro. Aparecerán en el terminal las opciones e instrucciones que guiarán paso a paso la introducción de todos los datos. Hay que prestar especial atención en el formato en el que hay que introducir los certificados, aunque la mejor opción es enviar los ficheros como texto a través del terminal, pues es muy fácil equivocarse si se envían escribiéndolos a mano.

Cuando ya está todo configurado, la tarjeta enviará cada 10 segundos la información captada por (casi) todos sus sensores en formato `.json` a AWS IoT. Para poder ver lo que envía hay que entrar en "Pruebas" desde la consola IoT Core y suscribirse al topic:

`$aws/things/`*<nombre del objeto>*`/shadow/update`

Cada mensaje que envía el nodo[4] es comprobado por la sombra (*shadow*) del objeto y reenviado de forma aumentada introduciendo *timestamp* a:

`$aws/things/`*<nombre del objeto>*`/shadow/update/accepted`

... si se introduce este último topic en "Pruebas" se podrá ver qué está devolviendo la sombra del dispositivo al recibir un mensaje desde el nodo.

Aún hay un par de cosas más que probar: si se pulsa en botón azul de la placa de desarrollo, el LED de pruebas LD2 de la PCB cambiará de encendido a apagado y viceversa, pero no lo está haciendo de forma interna al dispositivo, lo que hace es enviar un mensaje MQTT a AWS IoT con formato `.json` que contiene el estado actual y el deseado (si está encendido el estado deseado será apagado y al contrario). El nodo no solo publica en `../update` si no que, además, también está suscrito a `../update/accepted` con lo que, cuando le llegue el mensaje —devuelto por la sombra— de que ha de cambiar el estado del LED al deseado, interpretará la orden y cambiará el LED. Este cambio en el estado de LD2 puede realizarse también "a mano" escribiendo en el visor/panel de pruebas de IoT Core el mensaje al tema `../update/accepted` como si dicho mensaje lo enviara otro dispositivo y ver cómo cambia sin necesidad de pulsar de forma local el botón azul.

Con este último "truco" se demuestra que los dispositivos conectados a AWS IoT no sólo pueden enviar información, si no que con el mismo sistema de MQTT también pueden realizar acciones de forma remota. De hecho, no tiene por qué ser el mismo dispositivo, puede ser otro el que de la orden de encender o apagar un LED.

[4] 'nodo' es otra forma de referirse a la STM32L4 Discovery Kit.

## 3.4 El ESP32 Devkit C y AWS IoT

Otro ejercicio muy interesante es emplear un dispositivo de distinta naturaleza[5] y ampliamente extendido y realizar unas conexiones similares a la del dispositivo específicamente pensado para conectarse con AWS IoT de forma que se demuestre la flexibilidad y versatilidad tanto del dispositivo como de AWS IoT.

Se trata de realizar unas pruebas con el ESP32 Devkit C. A pesar de no necesitar presentación, indicar que se trata de una pequeña placa de desarrollo —y no sólo de desarrollo, pues también se usa en producto final— que posée un módulo WiFi, uno Bluetooth, tiras de pines configurables y programables y un sinfín de librerías y documentación para poder realizar experimentos electrónicos con el añadido de tener la pila TCP/IP integrada y poder conectar con mucha facilidad a una red WiFi y, por lo tanto, a Internet. Para demostrar la versatilidad, se emplearán dos plataformas de desarrollo con dos lenguajes de programación distintos para conseguir el mismo objetivo: conectar con AWS IoT:

- Entorno Zerynth y Python
- Entorno Arduino IDE y C/C++

### 3.4.1 Zerynth y Python

Para instalar Zerynth Studio hay que dirigirse a su web [29], descargar el instalador en función del sistema operativo en el que zse esté trabajando, Linux 64 bit para el caso concreto de este TFG y seguir los pasos de instalación. La instalación pedirá un registro que habrá que realizar si no se tiene ya una cuenta en este sistema.

Por no hacer demasiado densa la lectura, no se profundizará en el apartado la configuración y el flujo de trabajo de Zerynth Studio de cara a hacer que un ESP32 Devkit C se conecte a AWS IoT.

Una vez instalado y configurado para funcionar con un ESP32 es el momento de empezar con el código. Lo mejor es comenzar con un ejemplo que viene en el IDE. Se sigue el videotutorial de [30] donde explican de forma clara cómo ha de realizarse el proceso.

Lo más recomendable es hacer una copia de la plantilla de ejemplo mediante una clonación[6] del ejemplo a probar, de esta forma no se tocará en ningún momento la plantilla/ejemplo. Después hay que modificarlo para que contenga los certificados generados, compilarlo y grabarlo en el ESP32 para, finalmente, hacerlo correr.

El ejemplo elegido es el que se encuentra dentro de `AWS/IOT`. Se clona dándole el nombre deseado y, opcionalmente, una descripción. En este punto se puede apreciar en el IDE que el proyecto está compuesto por varios ficheros. Uno de ellos "`thing.conf.json`" ha de contener la configuración de la conexión a la plataforma AWS IoT. Se necesita también la dirección del *endpoint* que indica la consola de AWS IoT Core en: Consola/Settings → Custom endpoint.

Como en anteriores ocasiones, hay que crear una política AWS IoT (se llamará `Politica_Zerynth`) que permita al dispositivo el acceso al servicio. El nombre de esta política irá en el fichero `.json` del proyecto Zerynth sustituyendo "`policy_name`".

A continuación hay que enlazar un objeto[7] de AWS IoT al proyecto de Zerynth que se está creando. Para realizar esto hay dos caminos:

[5] Con "distinta naturaleza" quiere decir que ni lleva el mismo microcontrolador ni si quiera es del mismo fabricante, con lo que no comparten prácticamente nada.

[6] Aunque basado en Git, no es del todo necesario tenerlo instalado pues el programa ya realiza la clonación.

[7] Hay que recordar que un objeto/*thing* es la entidad de AWS IoT que representará de forma abstracta al dispositivo físico, el ESP32, en la nube.

- Con el *toolchain* de Zerynth: `ztc`
- Creando "a mano" el objeto, los certificados y escribiendo/rellenando el fichero `thing.conf.json`

La primera opción no parece llevarse bien con Linux Debian, con lo que se opta por la segunda que, aunque a priori parezca más engorrosa, resulta la más conveniente pues se hace con las herramientas que se han empleado hasta ahora.

En primer lugar ha de crearse el objeto y generar sus certificados como ya se ha visto en secciones anteriores. Los certificados hay que descargarlos en local pues habrá que añadirlos al proyecto de Zerynth. Se asocia el objeto a la política creada antes y se activan los certificados.

Dentro del fichero `thing.conf.json` está el campo "`cert_arn`". El valor a rellenar se encuentra en AWS IoT Core → Seguro → Certificados `Politica_Zerynth` → ARN de certificado. En los campos "`mqttid`" y "`thingname`" se escribirá el objeto creado.

Dentro de la carpeta del proyecto hay que añadir dos de los ficheros descargados desde AWS IoT:

- El certificado del objeto que habrá que guardarlo en la carpeta del proyecto como "`certificate.pem.crt`"
- La clave privada que tendrá el nombre "`private.pem.key`"
- Opcionalmente se podría añadir el CA pero, para ello, habrá que modificar dicho apatadao en el código. En esta ocasión no se hace.

El siguiente paso es configurar la conexión WiFi en el fichero "`main.py`" en torno a la línea 35 del código, donde se editará tanto la SSID como la contraseña de la red y su seguridad. Aquí, en este fichero, se pueden modificar más parámetros para que, por ejemplo, los datos que se envíen tengan algún significado que se desée o se pueden dejar tal cual está de tal forma que cada segundo, el dispositivo enviará a la nube un número al azar entre 0 y 9.

Se verifica el código (≡ compilar[8]) en el botón '*Verify*' y se sube (≡ grabar el dispositivo ESP32) con el botón '*Uplink*'. Es conveniente abrir el terminal serie de Zerynth para poder ver lo que está devolviendo el ESP32 conectado de forma local. Por otro lado, en el apartado de pruebas de AWS IoT, hay que suscribirse al *topic* donde el dispositivo envía los mensajes para poder observar cómo, lo que envía, tiene formato `.json`.

A través de este ejemplo, no sólo se puede comprobar cómo el dispositivo envía datos a AWS IoT, también a la inversa. ¿Cómo? Se puede cambiar el periodo/frecuencia con el que el ESP32 envía los datos a través de su sombra (*shadow*). Desde AWS IoT Core se accede al apartado: Administración → Objeto → <nombre del objeto>→ Sombra. Se edita el estado de la sombra sustituyendo "`reported`" por "`desired`" y escribiendo el período deseado en ms. Al guardar, aparecerá el mensaje de que se ha actualizado la sombra correctamente. Volviendo a la consola de pruebas, se puede comprobar con el *timestamp* que el cambio se ha efectuado pues ha cambiado la frecuencia con la que el dispositivo envía la información.

Como conclusión se puede comprobar que, como se indica en el apartado anterior, la información no sólo fluye del ESP32 a la nube AWS IoT, si no que también se pueden enviar datos, parámetros, comandos... desde la nube (es decir, desde cualquier punto, incluso otro dispositivo IoT) al dispositivo para modificar su comportamiento y que realice diversas acciones.

[8]Python es un lenguaje interpretado, no compilado. De ahí que haya que crear la capa de abstracción con el intérprete que se añadirá al código del proyecto.

### 3.4.2 Arduino IDE

Para esta sección se ha partido del TFG de Álvaro Benito Herranz[31]. Dicho TFG está basado en Windows, pero para la parte de las configuraciones y las ideas generales, sirve de referencia para este desarrollo en Linux.

Continuando con las diferentes formas de conectar un dispositivo sin importar su naturaleza o el entorno de desarrollo, se sigue el tutorial de la web *instructables*[32] y se comprueba cómo realizar una conexión a AWS IoT desde un dispositivo ya probado desde un entorno más sencillo como es el Arduino IDE y haciendo uso de todas las facilidades que nos proporciona tanto para generar código (tanto desde ejemplos como desde cero) como para configurar el dispositivo. Se usarán las librerías `Arduino-esp32-aws-iot` de *Hornbill*[9].

A pesar de que tiene muchos pasos, el procedimiento es sencillo pues ya se han realizado unos cuantos.

- Crear una cuenta en AWS. Ya se ha creado con anterioridad para realizar conexiones anteriores.
- Crear un objeto. Ídem.
- Generar y descargar certificados para el objeto creado. Ídem.
- Definir/crear una pilítica y asociarla al objeto creado. También realizado.
- Probar mediante la herramienta `MQTT fx` que se han realizado correctamente todos los pasos anteriores.
- Una vez el Arduino IDE[33] instalado hay que añadirle un gestor externo de tarjetas adicionales (menú Archivo/Preferencias) y así poder configurar el entorno de desarrollo para que reconozca, compile y escriba firmware desde la aplicación al ESP32.
- Instalar las librerías de *Hornbill* anteriormente mencionadas.
- Abrir el ejemplo de las librerías `AWS_IOT/pubsubTest` completando el código con los datos de la conexión Wifi y los certificados generados con anterioridad.
- Comprobación de que el ESP32 conecta y funciona de dos formas distintas:
  - Mediante el `Serial monitor` integrado en el Arduino IDE, donde el propio dispositivo va escribiendo el paso que está realizando y si lo ha realizado correctamente.
  - A través de la herramienta `MQTT fx` suscribiéndose al topic del objeto y/o al de su sombra.

En los últimos pasos del tutorial, se indica cómo realizar unas modificaciones al código y añadir unos pocos elementos al dispositivo *hardware* para que lo que envíe a AWS IoT no sea un contador si no, la temperatura y humedad medidas.

## 3.5 El entorno STM32 de ST Microelectronics

Una vez realizadas todas las aproximaciones anteriores, el desarrollo se enfoca en el entorno de *ST Microelectronics*, pues la finalidad es la de trabajar con la tarjeta de desarrollo *STM32L4 Discovery*

[9] En la dirección https://github.com/ExploreEmbedded/Hornbill-Examples/tree/master/arduino-esp32/AWS_IOT se pueden descargar [Último acceso septiembre de 2019] **Quizá sea mejor una cita bibliográfica**

*Kit*. Se decide por seguir los tutoriales del fablicante[10] para familiarizarse con las distintas aplicaciones y entornos de desarrollo (en definitiva: las herramientas de desarrollo) que tiene a disposición ST. Como es obvio, se eligen los más recomendados y para el sistema operativo Linux: *STM32CubeMX* de *ST Microelectronics* y *System Workbench for STM32* de *ac6*.

Siguiendo los cinco tutoriales (o *steps* como lo denomina el portal) propuestos por *ST* no solamente se consigue familiarizarse con los IDEs de desarrollo, también se obtiene un conocimiento adicional de dos de las PCBs de desarrollo más representativas del fabricante (una de ellas es la *STM32L4 Discovery Kit*) que se añade a la experiencia de las anteriores secciones. Los hitos están claramente definidos en cada *step* o tutorial:

- **Step 1**: Instalación, descripción básica y ejemplo de utilización de las diferentes herramientas para el desarrollo.
- **Step 2**: Se profundiza en las *librerías HAL* y se configuran los pines de I/O para realizar un sencillo programa *blink LED*.
- **Step 3**: Cómo se programa el uso de una UART en los kits de desarrollo.
- **Step 4**: Cómo configurar los periféricos para medir temperatura con el sensor integrado en *Discovery kit*. Este punto es muy importante porque será empleado más adelante en el desarrollo final junto a la medición de la humedad.
- **Step 5**: Muestra cómo comunicar en nodo *Discovery kit IoT* con un teléfono móvil mediante Bluetooth. En este tutorial se aprecia la versatilidad de esta placa de desarrollo.

Realizados los tutoriales se está en buena posición para afrontar los siguientes pasos.

## 3.6 STM32 y Amazon FreeRTOS

En la sección anterior se programa el *Discovery IoT node* de forma local para que realice las tareas que se desean, pero no se conecta más allá. En este apartado (y en el siguiente) ya se hace que el dispositivo se conecte a AWS IoT y transmita información. En una primera fase (esta sección) se descargará un ejemplo, se editará con las configuraciones necesarias, y se analizará qué hace y cómo. En la siguiente fase/sección se modificará el código para adaptarlo a las necesidades especificadas.

### 3.6.1 Obtención del ejemplo Amazon FreeRTOS para el kit

Desde la página principal de AWS se puede acceder a la "Guía del usuario de FreeRTOS" y de ahí –con algún salto intermedio– se llega a la Guía de introducción específica de la placa *STMicroelectronics STM32L4 Discovery Kit IoT Node*[11].

Los primeros pasos indican cómo preparar y configurar las cuentas, permisos, etc del entorno de AWS IoT y cómo descargarse FreeRTOS adaptado al dispositivo que tenemos entre manos. En cuanto a los ajustes de permisos y cuentas, no es más que lo realizado en secciones anteriores de este TFG pero, por otro lado, en este punto puede sonar un tanto ambiguo el concepto de "descargar FreeRTOS", pues bien: se refiere a descargar un proyecto para *System Workbench for STM32* con las librerías...

[10] Página de entrada a los tutoriales: https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-step-by-step.html?ecmp=tt8485_gl_link_oct2018 - [Último acceso: septiembre 2020]

[11] https://docs.aws.amazon.com/freertos/latest/userguide/getting_started_st.html - [Último acceso: septiembre de 2020]

- ... de bajo nivel para la *STM32 Discovery kit*
- ... de FreeRTOS para programar en tiempo real
- ... para conectarse vía internet con AWS IoT
- ... más un ejemplo de cómo se han de emplear el conjunto de todas las librerías anteriores y enviar –por MQTT– unos mensajes al *broker* de AWS IoT.

Esto quiere decir que **Amazon, al convertir FreeRTOS en 'su' Amazon FreeRTOS no sólo añade unas funciones para las comunicaciones seguras con AWS IoT, también está incluyendo toda la capa de comunicaciones entre dispositivo y *router* WiFi, añade también la implementación del protocolo TCP/IP y, de alguna forma, también fusiona las librerías propias del fabricante del dispositivo/kit abstrayendo (léase capa HAL) los accesos a cada uno de los dispositivos físicos implementados en la placa, con unas llamadas a funciones**.

Esto hace que se reduzcan mucho los tiempos de implementación en cuanto a las comunicaciones y adquisiciones de datos, así como la adaptación a las necesidades de cada desarrollo final, pues ya ofrece un "proyecto plantilla" con todos los ingredientes que se pueden necesitar para usar el *hardware* integrado en cada kit de desarrollo.

Nota del Autor: Antes de continuar al siguiente punto es importante observar las advertencias sobre la versión del *firmware* del dispositivo wifi integrado en el *Discovery kit node* pues, como le pasó al autor de este TFG, hubo que actualizarlo no sin complicaciones para que funcionase el ejemplo.

Volviendo a la guía de introducción y a la descarga, hay que navegar por varios *links* hasta llegar al enlace que descargue un fichero con el nombre:

```
Connect to AWS IoT-STM32-B-L475E-IOT01A
```

para la plataforma de *hardware*:

```
STM32L4 Discovery kit IoT node
```

Por otro lado, para poder abrir el proyecto y empezar a escribir código, hará falta el IDE *System Workbench for STM32* [25]. Para instalar el IDE solo hay que registrarse en la web y seguir unos sencillos pasos. Finalmente, en los últimos pasos del "*getting started*" indica cómo compilar el proyecto-ejemplo importado. Antes de esto, si no se desea ver como no funciona nada, hay que añadir unas cuantas configuraciones a diferentes *headers* del proyecto.

### 3.6.2 Editando *headers* del ejemplo

Como se indicaba en el punto anterior, para que todo funcione en el entorno en el que se está desarrollando, hay que indicarle unos cuantos parámetros al dispositivo. Parámetros como:

- SSID de la red WiFi a la que debe engancharse para salir a internet.
- La contraseña de dicha red WiFi.
- El tipo de encriptado.
- El *broker* al que ha de enviar los mensajes MQTT.
- La identificación del "Objeto IoT" que es para que lo reconozca AWS.

- El certificado de objeto y de usuario para las comunicaciones seguras.

Cada una de ellas se detalla a continuación:

### WiFi, broker y objeto

Estos tres parámetros se escriben en el mismo *header*. Desde la carpeta raíz del proyecto, se busca el fichero en:

`./demos/include/aws_clientcredential.h`

y, al abrirlo, se han de rellenar los siguientes `#define`:

- `#define clientcredentialWIFI_SSID`: Ha de escribirse el SSID de la Wifi entre comillas, por ejemplo: `"WLAN_ABCD"`
- `#define clientcredentialWIFI_PASSWORD`: Ha de escribirse la contraseña de la red WiFi entre comillas, por ejemplo: `"1234567890abcdef1234"`
- `#define clientcredentialWIFI_SECURITY`: El tipo de encriptación de la red WiFi, se puede dejar por defecto: `eWiFiSecurityWPA2` (sin comillas)
- `#define clientcredentialMQTT_BROKER_ENDPOINT`: La dirección del *broker*[12] de la cuenta asociada de AWS entre comillas, por ejemplo: `"plfitrmns13a1-txf.iot.us-east-2.amazonaws.com"`
- `#define clientcredentialIOT_THING_NAME`: El nombre del objeto que esté declarado en la consola de AWS IoT y que se va a conectar, por ejemplo: "STM32discoveryIoT"

Los siguientes es recomendable dejarlos por defecto:

- `#define clientcredentialMQTT_BROKER_PORT`
- `#define clientcredentialGREENGRASS_DISCOVERY_PORT`

### Los certificados

Estos certificados se escriben en el *header*:

`./demos/include/aws_clientcredential_keys.h`

Para que el Discovery kit node pueda comunicarse correctamente con AWS tendrá que disponer de los certificados que acrediten sus permisos de comunicación, esto se hace con la inclusión de solo dos certificados: el certificado del objeto[13] y la clave privada del usuario.

Para escribir ambos certificados (`#define keyCLIENT_CERTIFICATE_PEM` y `#define keyCLIENT_PRIVATE_KEY_PEM`) en el *header* hay que prestar especial atención, pues son cadenas bastante largas con letras, números y símbolos. Si se leen con atención los comentarios previos a cada uno, se ve que la forma de incluirlos ha de ser similar a:

[12] Esta dirección se obtiene dentro de la consola AWS IoT, en el apartado de "Configuración", punto de enlace
[13] No hay que olvidar activar el certificado o, de lo contrario, no funcionará.

```
#define keyCLIENT_CERTIFICATE_PEM   "-----BEGIN CERTIFICATE-----\n"\
"MSIDoTCyAklgAwIBTgIeTgrhcIrFMuefpMmdoCcvtQNdlONv/p0JKoZIhvcNAQEL\n"\
"yQAwTTFvLMEkGeA1UEnCwxCgQW1heom9udIFdleYiBTZXaJ2AWNllcyBPkPUFo0g\n"\
[...]
"ErX7Mlyh/czdqKVOB7o3CetoY1f008rAmlrgOIguqnyXHLyY35tOfyPm2Ag\n"\
"-----END CERTIFICATE-----\n"
```

Listado de código 3.3: Formato de escritura de certificados en *header*

### 3.6.3 Ejecutando el ejemplo

Una vez completados los puntos anteriores, se compila el ejemplo como se indica en la web y se ejecuta. Para poder ver lo que está enviando, en la consola AWS IoT, dentro del menú Test ("Prueba" si se tiene el idioma en español) hay que suscribirse al topic que indica la web: `iotdemo/#`. Hecho esto, se pueden ver los mensajes (en texto plano, no en formato `.json`) que está enviando la placa *STM32 Discovery IoT node* hasta parar.

Para tener aún más información de lo que está haciendo antes de analizar el código, se puede conectar un terminal serie al puerto serie virtual que crea el dispositivo al ser conectado. Un monitor de puerto serie tipo `CuteCom`[14] es suficiente para ver lo que envía el STM32 por el puerto virtual USB y dar una pista de los pasos que va siguiendo...

```
0 528 [Tmr Svc] WiFi module initialized.
1 1087 [Tmr Svc] Write certificate...
2 4677 [Tmr Svc] WiFi connected to AP WLAN_AAAA.
3 4681 [Tmr Svc] IP Address acquired 192.168.1.160.
4 4686 [Tmr Svc] WiFi firmware version is: C3.5.2.5.STM
5 4691 [Tmr Svc] WiFi firmware is up-to-date.
6 4696 [iot_thread] [INFO ][DEMO][4696] ----------STARTING DEMO----------
7 4703 [iot_thread] [INFO ][INIT][4703] SDK successfully initialized.
8 11334 [iot_thread] [INFO ][DEMO][11334] Successfully initialized the demo. Network ty[...]
9 11343 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:745]
10 11351 [iot_thread] Creating a TLS connection to aivjcbhlx43wb-ats.iot.us-east-2.amaz[...]
12 13005 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:512]
13 13012 [iot_thread] Creating an MQTT connection to aivjcbhlx43wb-ats.iot.us-east-2.am[...]
15 13193 [iot_thread] [INFO] [MQTT] [core_mqtt.c:886]
16 13198 [iot_thread] Packet received. ReceivedBytes=2. [...]
18 13205 [iot_thread] [INFO] [MQTT] [core_mqtt_serializer.c:970]
19 13211 [iot_thread] CONNACK session present bit not set. [...]
21 13218 [iot_thread] [INFO] [MQTT] [core_mqtt_serializer.c:912]
22 13224 [iot_thread] Connection accepted. [...]
24 13230 [iot_thread] [INFO] [MQTT] [core_mqtt.c:1563]
25 13235 [iot_thread] Received MQTT CONNACK successfully from broker. [...]
27 13243 [iot_thread] [INFO] [MQTT] [core_mqtt.c:1829]
28 13248 [iot_thread] MQTT connection established with the broker. [...]
30 13257 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:820]
31 13264 [iot_thread] An MQTT connection is established with aivjcbhlx43wb-ats.iot.us-e[...]
```

[14] Configuración: /dev/ttyACM0 115200-8-N-1.

```
33 13276 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:885]
34 13283 [iot_thread] Attempt to subscribe to the MQTT topic STM32discoveryIoT/example/[...]
36 13296 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:899]
37 13303 [iot_thread] SUBSCRIBE sent for topic STM32discoveryIoT/example/topic to broke[...]
39 13437 [iot_thread] [INFO] [MQTT] [core_mqtt.c:886]
40 13442 [iot_thread] Packet received. ReceivedBytes=3. [...]
42 13449 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:1053]
43 13456 [iot_thread] Subscribed to the topic STM32discoveryIoT/example/topic with maxi[...]
45 14467 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:533]
46 14474 [iot_thread] Publish to the MQTT topic STM32discoveryIoT/example/topic. [...]
48 14489 [iot_thread] [INFO] [MQTT_MutualAuth_Demo] [mqtt_demo_mutual_auth.c:543]
49 14496 [iot_thread] Attempt to receive publish message from broker. [...]
```

Listado de código 3.4: Salida por el puerto virtual

Con toda esta información, lo que se puede ver en la consola MQTT de AWS IoT y siguiendo con atención el código, el ejemplo realiza las siguientes operaciones:

1. El sistema inicializa la conexión WiFi registrándose en la red. Después de todo, si no hay conexión a internet, no servirá de nada el resto.
2. Una vez creada la conexión, crea varios hilos:
   - *thread* [INIT]: que pertenece al sistema y controla las librerías que se cargan, descargan, inicio y fin.
   - *thread* [MQTT]: que se encarga de gestionar tanto las publicaciones vía MQTT como los mensajes que llegarán por suscripción.
   - *thread* [DEMO]: que pertenece al marco en el que se encuadra el ejemplo en ejecución haciendo las inicializaciones necesarias.
   - *thread* [MQTT_MutualAuth_Demo]: son las llamadas a funciones propias del ejemplo.
3. Inicia las librerías que se van a usar.
4. Repetirá tres veces la siguiente secuencia:
   - Conecta con el *broker* identificándose con los valores introducidos previamente.
   - Se suscribe al mismo *topic* en el que va a publicar para confirmar que se ha enviado y el *broker* lo ha publicado.
   - Envía cinco mensajes MQTT esperando y mostrando la confirmación de que se han publicado en el *broker*.
   - Se des-suscribe del *topic*.
   - Desconecta del *broker*.
5. En paralelo, el *hilo* [MQTT] va encargándose, de forma asíncrona, de enviar y recibir los mensajes intercambiados con el *broker* (de publicaciones y recepción de mensajes de suscripciones) y enviando por el puerto serie las acciones realizadas.

Con este ejemplo se pueden ver el grueso de acciones con las que trabajar con Un dispositivo basado en microcontrolador y conectado a AWS IoT con MQTT. Se tienen por un lado las configuraciones y certificados de AWS, por otro lado la conexión a WiFi y, desde el punto de vista del microcontrolador

con Amazon FreeRTOS, la secuencia de acciones a tomar previas al envío (publicación) de datos y cómo procesar tanto los estados del envío como la suscripción a *topics* y la recepción de mensajes de dichas suscripciones.

### 3.6.4 Estructura del ejemplo

Gracias a que muestra todas las operaciones básicas a realizar y estructura muy bien el orden, el ejemplo anterior sirve como plantilla base para conseguir el objetivo final del TFG. Para ello, antes hay que profundizar un poco más en la estructura de las funciones del código y poder elegir de forma más óptima dónde y de qué forma añadir el código que haga que se envíen la temperatura y la humedad en un mensaje en formato `.json`.

Cabe recordar y destacar que Amazon (o quizá el propio fabricante del *hardware*) ya se ha encargado de abstraer el código de los dispositivos físicos, así como la conexión al punto WiFi, facilitando enormemente el desarrollo de una aplicación como la que concierne a este documento. Quiere esto decir que serán obviadas las explicaciones de dichas capas.

En la Figura 3.1 se puede ver, de forma esquemática, la estructura (o mejor: infraestructura) que establece Amazon FreeRTOS antes de llegar a la llamada de la función `RunCoreMqttMutualAuthDemo()`.

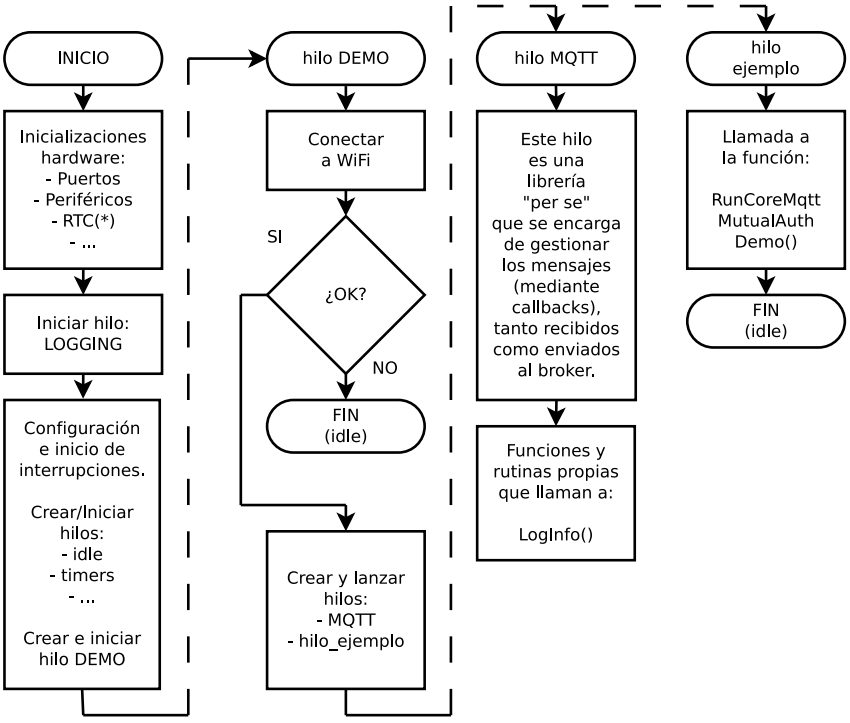


Figura 3.1: Diagrama del ejemplo Amazon FreeRTOS

(*) Inicializa el *hardware* RTC pero no lo pone en hora actualizada.

Dicha función, que se halla dentro del fuente `mqtt_demo_mutual_auth.c`, es donde se encuentra el procedimiento mencionado al final del apartado anterior. En la Figura 3.2 se puede ver un diagrama de bloques del funcionamiento.

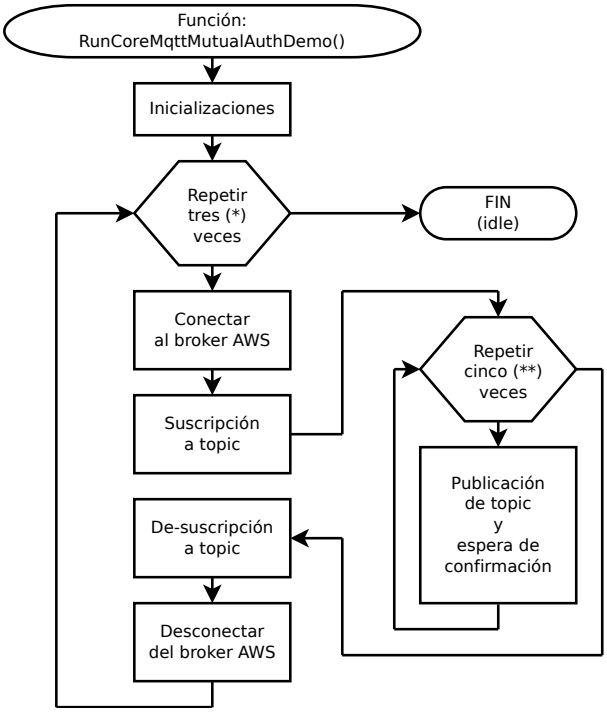


Figura 3.2: Función `RunCoreMqttMutualAuthDemo()`

(*) En realidad no son tres, es la macro `democonfigMQTT_MAX_DEMO_COUNT`.

(**) En realidad no son cinco, es la variable `ulMaxPublishCount`.

### 3.6.5 Primeros pasos en las mediciones de temperatura y humedad

Ya se ha visto como funciona el ejemplo que muestra un sistema cuasi-completo de publicación y suscripción de mensajes, así como de todos los pasos previos que conlleva. También se ha llegado a la función `RunCoreMqttMutualAuthDemo()` y se ha descrito el guion que ejecuta paso a paso.

Pudiera parecer que este código hiciera todo y el ingente resto de ficheros con miles de líneas de código no hiciera nada. De hecho, llegado este punto, y teniendo el conocimiento de las herramientas que ofrece ST junto con los IDE, cabría preguntarse ¿por qué no crear un proyecto de cero con la herramienta *STM32CubeMX* en lugar de modificar el ejemplo? La opción de empezar de cero daría la sensación de crear un proyecto límpio, faltaría la capa de conexión a AWS IoT, pero se podría añadir con posterioridad. Parece sencillo.

Nada más lejos de la realidad: este *source* actúa más bien como un *frontend*, como un *script*, que ordena realizar tareas mucho más complejas con "solo" llamar a unas pocas funciones. El grueso de la algoritmia está en esa masa infinita de librerías. Ni que decir tiene que, además, por debajo de todo ello está FreeRTOS que actúa como sistema operativo pudiendo crear diferentes *hilos* de ejecución y gestionando la memoria para que nada se escape. La complejidad que acarrearía "hacerlo a mano" y la facilidad de que se omita un paso a la hora de integrarlo todo, daría muchos quebraderos de cabeza y restaría un tiempo muy valioso de diseño.

En definitiva, será esta función (y el fichero fuente que la contiene) el punto de entrada elegido para ser modificado y adaptado a los requerimientos del presente TFG. Se editará para que tome las medidas de temperatura y humedad que entrega el senstor integrado en el *STM32 Discovery IoT node*, genere el mensaje y lo envíe por MQTT en formato `.json` para que, después, se pueda procesar del lado de AWS IoT y de un suscriptor externo que mostrará gráficamente la evolución de las magnitudes medidas.

### 3.6.6 Las funciones de toma de medidas y generación del formato elegido

Si se busca un poco en las ramas del proyecto, entre todas las librerías, también aparecen unos ficheros (*header + source*) que son las funciones de bajo nivel que proporcionan el control de todos los sensores y dispositivos que tiene la placa de desarrollo.

`./vendors/st/stm32l475_discovery/BSP`

En esta carpeta y subcarpetas se encuentran las declaraciones de funciones (y su implementación) que ayudarán a desarrollar el proyecto que se desea de forma más sencilla que si hubiera que escribir todo el código de bajo nivel para comunicar con los periféricos.

Siguiendo con la idea original de medir temperatura y humedad, se buscan las funciones que devuelvan estas medidas del sensor HTS221 que tiene integrado la placa. En los *header* `stm32l475e_iot01_tsensor.h` y `stm32l475e_iot01_hsensor.h` (mencionados en el Listado 2.2) se encuentran los prototipos de las funciones para la temperatura y humedad relativa.

```
uint32_t BSP_TSENSOR_Init(void);
float    BSP_TSENSOR_ReadTemp(void);
```

Listado de código 3.5: Prototipos de funciones para medida de temperatura

```
uint32_t BSP_HSENSOR_Init(void);
float    BSP_HSENSOR_ReadHumidity(void);
```

Listado de código 3.6: Prototipos de funciones para medida de humedad

Como se puede observar, los sensores tienen una función "`Init`" que hay que llamar antes de tomar la medida en sí. Es un detalle de apariencia insignificante pero, de olvidarse, la función de lectura no devolverá un valor válido. Dichas llamadas de inicialización, se realizarán al principio de la función de entrada, justo después de declarar todas las variables de la misma.

```
    BSP_TSENSOR_Init();
    BSP_HSENSOR_Init();
```

Listado de código 3.7: Llamadas de inicialización

El ejemplo de partida publica mensajes MQTT en texto plano que el *broker* representa como una cadena UTF-8 sin más. La idea original es que los mensajes tengan un formato `.json` para que tengan una estructura ordenada y sean más sencillos de almacenar en una base de datos. Para adaptar este punto hay que crear una pequeña función que genere una cadena de caracteres (en dicho formato `.json`) que será el mensaje a transmitir. La primera idea sería la que se muestra en el Listado 3.8.

```
1  int generaJson(char * cadena, uint32_t indice) {
2      float   humedad = BSP_HSENSOR_ReadHumidity(),
3              temperatura = BSP_TSENSOR_ReadTemp();
4      int     long_cadena;
5
6      long_cadena = snprintf(cadena, MAX_JSON_GENERADO,
7                             "{ \"ind\": %d, \"humedad\": %.2f, \"temperatura\": %.2f }\n",
8                             indice, humedad, temperatura);
9      return long_cadena;
10 }
```

Listado de código 3.8: Función generadora de .json –primera idea–

... pero el compilador no soporta el modificador de formato %f dentro de la función snprintf() con lo que habrá que convertir cada número flotante en dos enteros y la función transformarse en la mostrada en el Listado 3.9.

```
1  int generaJson(char * cadena, uint32_t indice) {
2      float   humedad = BSP_HSENSOR_ReadHumidity(),
3              temperatura = BSP_TSENSOR_ReadTemp();
4      int     hrInt1, hrInt2, tmpInt1, tmpInt2;
5      float   valFrac;
6      int     long_cadena;
7
8      hrInt1  = humedad;
9      valFrac = humedad - hrInt1;
10     hrInt2  = trunc(valFrac * 100);
11     tmpInt1 = temperatura, tmpInt2;
12     valFrac = temperatura - tmpInt1;
13     tmpInt2 = trunc(valFrac * 100);
14
15     long_cadena = snprintf(cadena, MAX_JSON_GENERADO,
16                            "{ \"ind\": %d, \"humedad\": %d.%02d, \"temperatura\": %d.%02d
                                   }\n"
17                            ,indice, hrInt1, hrInt2, tmpInt1, tmpInt2);
18
19     return long_cadena;
20 }
```

Listado de código 3.9: Función generadora de .json –idea final–

Con la llamada a esta última función (pasándole un array de char) se obtiene una cadena como la del Listado 3.10 que ya está en formato .json para que el *broker* y el entorno AWS IoT lo tome como tal. Se añade un índice al principio para poder distinguir un mensaje de otro. En este caso es un contador pero, más adelante, se cambiará por un índice temporal unívoco.

```
{ "ind": 4257, "humedad": 42.38, "temperatura": 29.87 }
```

Listado de código 3.10: Resultado de generaJson()

El valor de MAX_JSON_GENERADO será una macro que lo fijará en 64 pues, si se cuentan los caracteres que conforman el mensaje .json, son 55 pero, pensando un par de pasos más adelante, este formato va a cambiar y en el índice puede haber un número que proceda de un entero de 32 bits; el valor máximo de la humedad será del 100.00 %; la temperatura pudede, en teoría, pasar de 100°C (aunque dado este caso, con toda probabilidad haya otros problemas mayores); al final de la cadena tiene que haber un byte terminador (0x0)... el valor de 64 parece bastante correcto como tamaño máximo –en caracteres– del mensaje.

Al tener que reformular la idea inicial de esta función, se hace necesario incluir la librería math.h para la funciión de truncado. Además, como se indica al principio de este apartado, en el inicio del *source* hay que incluir las referencias a los headers que darán acceso a las funciones/periféricos de medida.

```
#include "stm32l475e_iot01.h"
#include "stm32l475e_iot01_tsensor.h"
```

```
#include "stm32l475e_iot01_hsensor.h"
#include <math.h>
```

Listado de código 3.11: Inclusión de *headers* para generar el `.json`

La función no solo rellenará el array de caracteres pasado por referencia, también devolverá el tamaño del mismo sin contar el `0x0`. Esto será útil para pasarle a la función que envía el mensaje MQTT el parámetro que lo requiere.

Siguiente paso ¿dónde llamar a la función generadora de `.json`?

Dentro de la función que se ha determinado como punto de entrada, en la fase de publicación de los topic, se llama a la función `prvMQTTPublishToTopic()` que es la encargada de crear e iniciar las estructuras necesarias (léase configuraciones) para proceder al envío en si de un mensaje que no dejará de ser texto plano en este punto (ya se ancargarán las librerías de "más abajo" de encriptarlo con una capa segura). Será en esta función donde haya que incluir a la función generadora, modificando un poco también las estructuras de la misma. Se sustituirán las primeras líneas de esta función por el código del Listado 3.12.

Así, cuando la función principal llame a esta para enviar los mensajes MQTT, en lugar de enviar el texto plano que enviaba en un primer momento, ya estará enviando un mensaje en formato `.json` que contiene una carga de información útil como es la temperatura y la humedad que detectan los sensores de la placa de ST. El índice que acompaña a esta información, no aporta mucho, es para distinguir un mensaje de otro en el caso (por ejemplo) de que las lecturas de temperatura y humedad sean iguales. Más adelante se profundizará en este índice y se convertirá en algo más útil.

```
1  static BaseType_t prvMQTTPublishToTopic( MQTTContext_t * pxMQTTContext ) {
2      MQTTStatus_t xResult;
3      MQTTPublishInfo_t xMQTTPublishInfo;
4      BaseType_t xStatus = pdPASS;
5      static int contador = 0;
6
7      /// tfg
8      char jsonGenerado[MAX_JSON_GENERADO] = { 0 };
9
10     /// Algunos campos no se usan y es mejor iniciarlos todos a cero
11     ( void ) memset( ( void * ) &xMQTTPublishInfo, 0x00, sizeof( xMQTTPublishInfo ) );
12
13     /// tfg
14     generaJson(jsonGenerado, (uint32_t) contador);
15     contador++;
16
17     /// El ejemplo usa QoS1
18     xMQTTPublishInfo.qos             = MQTTQoS1;
19     xMQTTPublishInfo.retain          = false;
20     xMQTTPublishInfo.pTopicName      = mqttexampleTOPIC;
21     xMQTTPublishInfo.topicNameLength = ( uint16_t ) strlen( mqttexampleTOPIC );
22     xMQTTPublishInfo.pPayload        = jsonGenerado;
23     xMQTTPublishInfo.payloadLength   = strlen( jsonGenerado );
24
25 [...]
26
27     return xStatus;
28 }
```

Listado de código 3.12: Modificación de la función `prvMQTTPublishToTopic()`

### 3.6.7 Modificación de etiquetas

Para poder ver los mensajes que envía *la placa ST* a AWS IoT, habría que entrar en el "Cliente de prueba de MQTT" de AWS IoT Core y suscribirse a un *topic* que equivale al nombre del objeto creado. para aligerar la carga visual de la monitorización, se opta por que el topic raíz será "`tfg/#`". Para ello, en el fuente `mqtt_demo_mutual_auth.c`, se sustituye siguiente macro por:

```
#define democonfigCLIENT_IDENTIFIER "tfg"
```

De esta forma, todos los mensajes enviados al *broker* AWS tendrán esa raíz de *topic*. Del mismo modo, se cambiará también el topic donde se enviarán las mediciones:

```
#define mqttexampleTOPIC democonfigCLIENT_IDENTIFIER "/discoveryIoT/hryt"
```

... pasando a ser "`tfg/discoveryIoT/hryt`".

N. de Autor: Debido a su extensión, el fichero `mqtt_demo_mutual_auth.c` modificado no se incluye en ningún apéndice de la memoria, se incluirá adjunto a este documento en su formato digital. Dicho fichero contiene en su interior comentarios para poder seguirlo fácilmente en los puntos donde no lo deje bien claro el presente documento.

### 3.6.8 Final de la primera modificación

Se llega aquí a un punto donde todo funciona con corrección. Se puede compilar, programar a la "placa ST" y todo funcionará fluirá siempre y cuando se pueda establecer una conexión WiFi y estén todas las *keys* bien escritas sin error. También han de estar correctamente configurados en AWS IoT el objeto/*thing* y las políticas. Si todo ello está correcto, solo hay que suscribirse al *topic* "`tfg/#`" del "Cliente de prueba de MQTT" de AWS IoT Core y se verá cómo, tras un tiempo inicial en el que establece enlace con el punto WiFi y se conecta a AWS, comienza a enviar los mensajes en el formato configurado. Opcionalmente, se puede obtener más información de lo que está haciendo el *Discovery kit* conectando un terminal al puerto serie virtual que crea en el PC (115200-8-N-1). Pero este trabajo está pidiendo un par de mejoras.

## 3.7 Mejora del código

Con anterioridad se pasó muy por encima sobre un índice que haría de marca diferenciadora entre mensajes, pero no era más que un contador que se incrementaba a medida que se enviaba un mensaje nuevo. Si se reiniciase de un modo u otro la ejecución del programa, este índice volvería a empezar de nuevo Esto significa que, si se quisiera almacenar en una base de datos, por ejemplo todos y cada uno de los mensajes que envía el *Discovery kit*, habría índices repetidos. ¿Cómo solucionar este problema?

### 3.7.1 Índice temporal en los mensajes

Dado que no se toma más de una muestra por segundo, con un índice temporal equivalente a segundos pareciera bastar, pero ¿segundos del día? ¿de la semana? ¿del mes? Suponiendo que el dispositivo se instalase en un sitio para recabar la evolución de la temperatura y la humedad durante un año, los índices se volverían a repetir. La solución que se adopta es que el índice será el "Tiempo Unix"[15].

[15] https://es.wikipedia.org/wiki/Tiempo_Unix

Adoptando que el índice sean los "segundos Unix", no solo se consigue que en todo momento el indice de cada mensaje sea diferente al de otro, si no que, además, aporta información sobre la hora UTC a la que se tomaron las mediciones. Esta última característica podría servir, por ejemplo, para odenar las medidas en el caso de que llegasen desordenadas a la base de datos. También se añade como ventaja la sencillez de que se trata de un número entero de 32 bit. No se hace necesaria una estructura *timestamp* ni similar.

Por otro lado, se vió que el código inicial arrancaba un hardware RTC al que no le ponía una hora real, con lo que, además, ya se tiene el soporte para que el dispositivo se ponga en hora al arrancar y no necesite más sincronizaciones de reloj[16], en todo momento el dispositivo tendrá la hora a la que se tomaron los valores de temperatura y humedad.

### 3.7.2 AWS *Lambda*

Parece buena idea, pero ¿cómo obtener la hora actual? Parece lógico pensar en el protocolo NTP[17] pero no hay una librería fácilmente asequible, con lo que habría que implementarla. Además, tampoco se hace necesaria la precisión que ofrece el protocolo. Otra alternativa a priori más sencilla, es emplear AWS Lambda[18], que son trozos de código (típicamente una función) que se ejecuta automáticamente cuando se cumplen las condiciones de disparo de la misma. Uno de los lenguajes de programación en los que se pueden escribir estas funciones Lambda es Python, con toda su versatilidad y flexibilidad.

La idea final es la siguiente: una vez –el dispositivo– ha arrancado y ha conectado por primera vez al *broker* AWS, se suscribe a un *topic* de respuesta y envía un mensaje que hace que se desencadene la ejecución de la función Lambda. Dicha función obtiene la hora UTC y la envía una respuesta `.json` al *topic* de respuesta al que se suscribió antes el objeto IoT. El mensaje contendrá la hora UTC y la hora Unix para que pueda procesarse más fácilmente al ser recibido. El *Discovery kit* lo captura y actualiza su RTC interno. Se de-suscribe del topic de respuesta y comienza a realizar todo el proceso de envío de temperatura y humedad capturados como al final de la Sección 3.6, con la diferencia que, ahora, cada vez que mide temperatura y humedad, captura también la hora y la convierte en hora Unix añadiéndola como índice, que pasa a tener peso específico y aportar mucha información.

Para crear la función Lambda sólo hay que entrar en el servicio AWS Lambda y pinchar en "Crear una función". Siguiendo los pasos solo hay que acordarse de elegir que el lenguaje sea Python 3.7 y que la "Instrucción de consulta de regla", es decir, la condición desencadenante de la ejecución, sea:

```
SELECT * FROM 'tfg/solicitud/sincrot'
```

¿Qué hace esta regla? Que cualquier mensaje enviado a ese *topic* desencadene la ejecución del código. ¿Qué código? el mostrado en el Apéndice A.

Por otro lado, obviamente, en el código del ST habrá que cambiar cómo se trata el mensaje recibido por parte de la función Lambda en el *topic* indicado para poder poner en hora el RTC interno. También habrá que crear unas funciones para actualizar y leer la hora del RTC que estarán en una pareja *header-source* llamados `tfgRTC.c/.h`. El código de estos fuentes está en el Apéndice B.

No hay que olvidar que, para realizar toda esta funcionalidad, hay que intercambiar en el esquema de la Figura 3.2 un bloque de petición de hora, tal y como se muestra en la Figura 3.3

[16] Ver Sección **??** Líneas futuras y mejoras.
[17] https://es.wikipedia.org/wiki/Network_Time_Protocol
[18] https://aws.amazon.com/es/lambda/features/

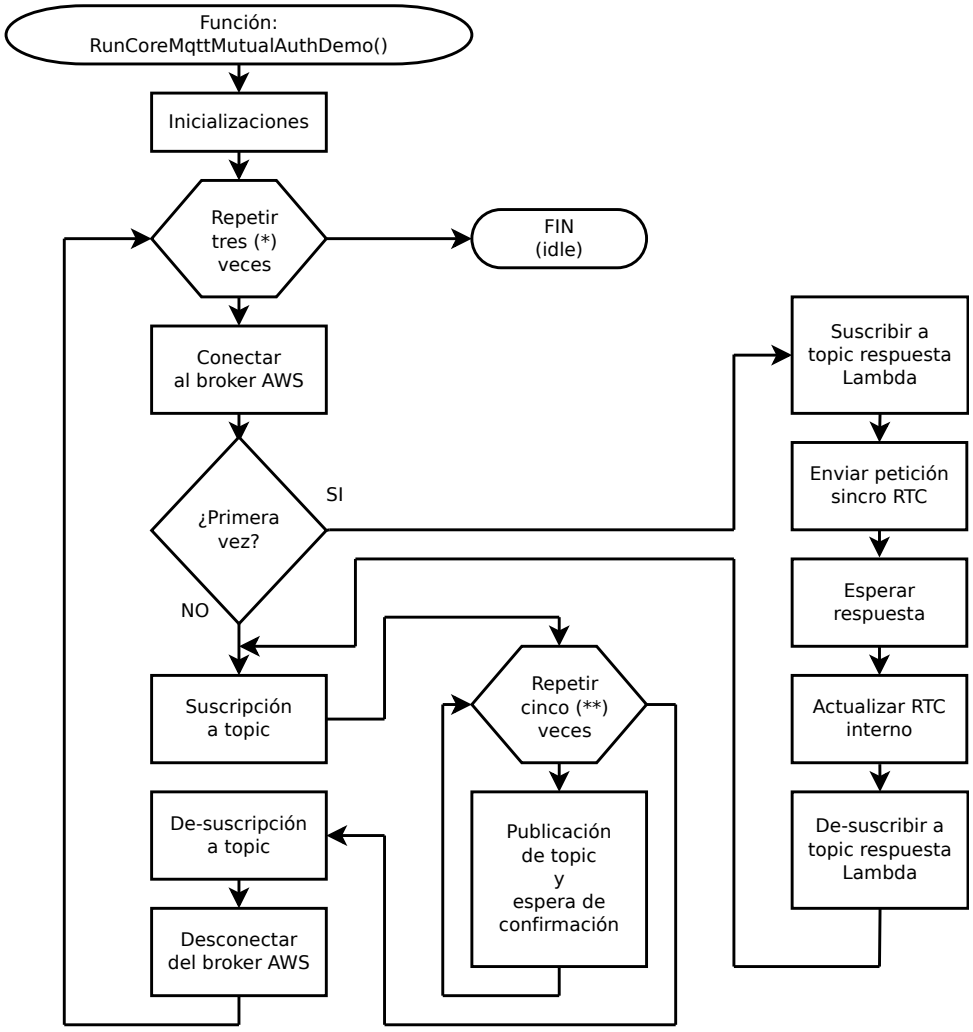


Figura 3.3: Función `RunCoreMqttMutualAuthDemo()` final

La función generadora de mensajes `.json` también ha de ser modificada para que, ahora, en lugar de poner como índice un contador que se incrementa, coja la hora del sistema, lo convierta en hora Unix y lo añada como índice. Puede verse cómoqueda finalmente en el listado de código 3.13

```
1  static int generaJson(char * cadena, uint32_t indice) {
2      float   humedad = BSP_HSENSOR_ReadHumidity(),
3              temperatura = BSP_TSENSOR_ReadTemp();
4      int     hrInt1 = humedad;
5      float   valFrac = humedad - hrInt1;
6      int     hrInt2 = trunc(valFrac * 100);
7      int     tmpInt1 = temperatura, tmpInt2;
8      int     long_cadena;
9      RTC_TimeTypeDef     sTime1;
10     RTC_DateTypeDef     sDate1;
11     HAL_StatusTypeDef   resultadoHAL_RTC;
12     tStruMsjJSON        fechaHora;
13     uint32_t            segundosUNIX;
14
15     valFrac = temperatura - tmpInt1;
16     tmpInt2= trunc(valFrac * 100);
17
18     resultadoHAL_RTC = HAL_RTC_GetTime(&xHrtc, &sTime1, RTC_FORMAT_BIN);
19     resultadoHAL_RTC = HAL_RTC_GetDate(&xHrtc, &sDate1, RTC_FORMAT_BIN);
```

```
20
21      fechaHora.horaUNIX    = 0;
22      fechaHora.segundoanyo = 0;
23      fechaHora.segundo     = sTime1.Seconds;
24      fechaHora.minuto      = sTime1.Minutes;
25      fechaHora.horaUTC     = sTime1.Hours;
26      fechaHora.dia         = sDate1.Date;
27      fechaHora.mes         = sDate1.Month;
28      fechaHora.anyo        = sDate1.Year + 2000;
29
30      convierteStruJSONaHoraUNIX(&fechaHora, &segundosUNIX);
31
32      long_cadena = snprintf(cadena, MAX_JSON_GENERADO,
33                             "{ \"ind\": %d, \"humedad\": %d.%02d, \"temperatura\":
34                             %d.%02d }\n",
35                             (int) segundosUNIX, hrInt1, hrInt2, tmpInt1, tmpInt2);
36
37      return long_cadena;
38  }
```

Listado de código 3.13: Función generadora de `.json` con *timestamp*

Finalmente, como se podrá ver en el fuente `mqtt_demo_mutual_auth.c` (que se adjunta digitalmente a esta memoria), también habrá que adaptar unas funciones a los últimos cambios:

- Función `prvMQTTProcessIncomingPublish()`: Se trata de la función de *callback* para cuando se recibe un mensaje por MQTT. Se adapta para que, cuando el mensaje recibido es de la respuesta de la función *Lambda*, analice el `.json` y actualice la hora del RTC interno.
- Se crean las macros `mqttPeticionUTCtopic` y `mqttRespuestaUTCtopic` con los *topics* que se han añadido para que las siguientes funciones corran sin problemas.
- Función `prvMQTTSubscribeWithBackoffRetries()`: Modificada para que puede suscribirse a los nuevos *topics* a tratar.
- Función `prvMQTTUnsubscribeFromTopic()`: Ídem que la anterior pero para de-suscribirse.
- Función `publicarPeticionUTC()`: Se añade esta función que crea un mensaje sencillo para enviar al topic que desencadena la ejecución de la función *Lambda*.

### 3.7.3 Final de la mejora

Después de las últimas modificaciones, el código ya no solo es funcional, si no que aporta mucha más información en práctimente el mismo tamaño de mensaje. De nuevo habrá que suscribirse al *topic* "`tfg/#`" del "Cliente de prueba de MQTT" de AWS IoT Core para poder ver los mensajes enviados. En esta ocasión también se podrán ver los que se intercambian para la actualización del RTC. La pega es que los números así presentados, no se asimilan tan bien como puede ser mediante un gráfico que los represente con respecto al tiempo. Pero eso se tratará en la siguiente sección.

## 3.8 Visualización de datos

Tal y como se introdujo en la Sección 2.3.5, no hay (a día de la escritura de esta memoria) un servicio sencillo que represente en forma de gráficas los datos enviados de temperatura y humedad. Lo que si

ofrece AWS son librerías para que, un usuario pueda comunicar con el *broker* de forma externa mediante una aplicación. La opción elegida es una aplicación escrita en Python para PC.

Tras leer la documentación ofrecida por AWS para comunicar con el *broker* usando Python 3, hay que instalar las librerías que tiene el mismo AWS para poder incluirlas en el script (como se muestra en el Listado de código 3.14). También serán necesarios los certificados generados en formato texto plano y almacenarlos en una carpeta que cuelgue de la que está el *script*.

```
from awscrt import io, mqtt, auth, http
from awsiot import mqtt_connection_builder
```

Listado de código 3.14: Librerías de AWS a incluir en Python

Sin entrar en detalle, pues no es el cometido de esta memoria, el *script* que contiene la interfaz gráfica está en el Apéndice C por si se quiere analizar. De forma somera, el *script* lanza dos hilos:

- Hilo principal: contiene la parte gráfica, la ventana que se muestra en pantalla. Cada cierto tiempo comprueba si hay valores en una cola de datos recibidos y los muestra.
- Hilo de conexión a AWS: este hilo se conecta al *broker* de AWS IoT y se suscribe a los topic donde publicará datos la *Discovery Kit*. El hilo a su vez define funciones *callback* para manejar los eventos de publicación de un mensaje en el/los *topic/s* suscrito/s, tratamiento de conexión interrumpida...

Mencionar que el *script* tiene un modo de funcionamiento test que permite comprobar que funciona la parte gráfica sin necesidad de comunicar con AWS. Para ejecutar el *script*:

```
$ python3 interfaztfg.py
```

Listado de código 3.15: Ejecutar interfaz en modo normal

```
$ python3 interfaztfg.py test
```

Listado de código 3.16: Ejecutar interfaz en moto *test*

El resultado final se muestra en la Figura 3.4. En una doble gráfica se puede ver la evolución de la temperatura y de la humedad en un periodo de tiempo. En la parte inferior se pueden apreciar los mensajes enviados por el dispositivo "traducidos" a texto. Se trata de una interfaz sencilla pero muestra con muy pocos elementos que todo funciona.

## 3.9 Almacenamiento de datos

Almacenar los mensajes enviados por los *edge devices* en una tabla de *DynamoDB* (ya se mencionó en la Sección 2.3.6) es sencillo como se puede comprobar en [34]. Para ello, es necesario:

- Crear una tabla *DynamoDB*. Desde la consola de DynamoDB se accede al botón "Crear tabla", tal y como se puede ver en la Figura **??**. Hay que dar un nombre a la tabla. En la clave principal:
  - Clave de partición: Asignar un nombre para el campo llave. En este caso la idea principal es que sea la hora Unix, pues tendrá un valor no repetible para este objeto.

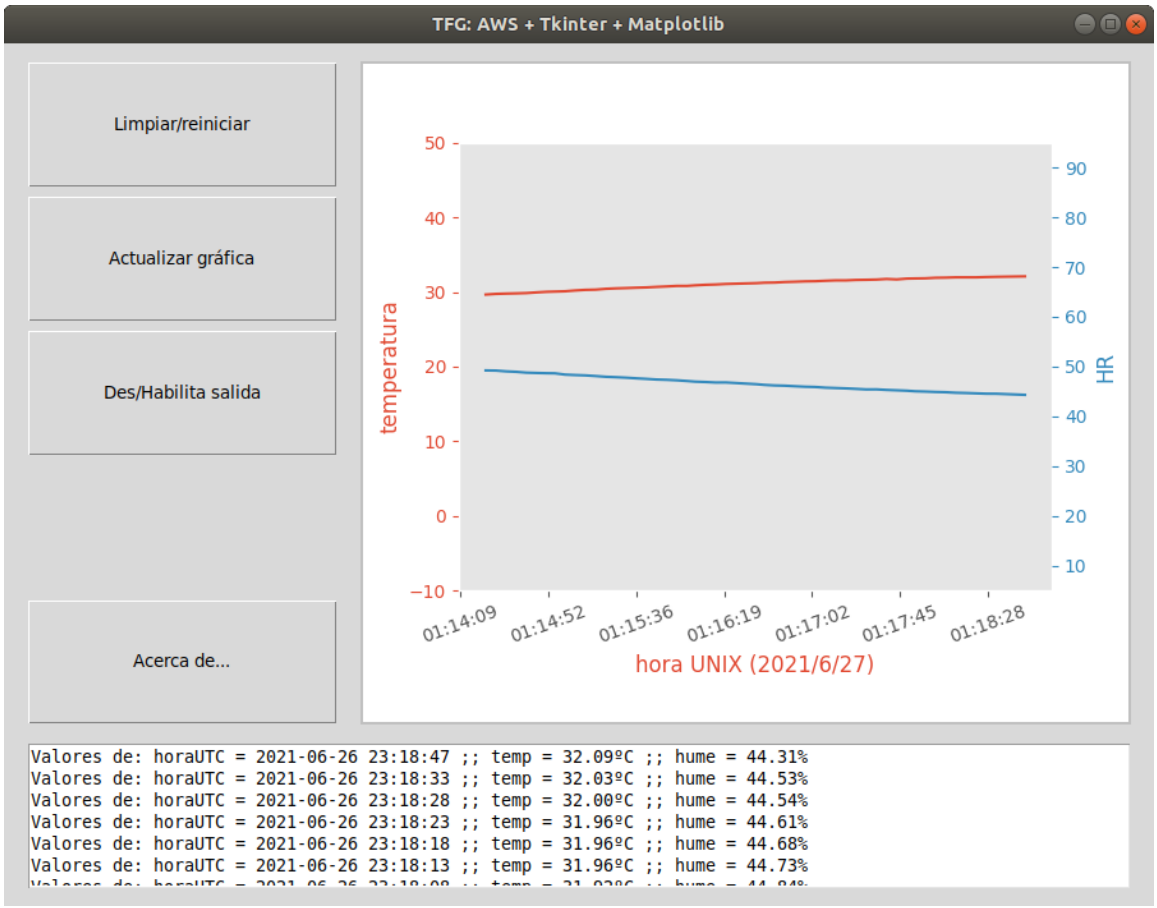


Figura 3.4: Interfaz gráfico en Python 3

  - Lista de opciones: “Number”
  - Añadir clave de ordenación: Hay que marcarlo para que se active. Este campo contendrá el nombre del dispositivo (en el caso de este trabajo sólo estará la *Discovery kit*, pero se podrían almacenar mensajes de más dispositivo publicados en el mismo *topic*).
  - Lista de opopciones de la clave de ordenación: “Number”

  Una vez se hayan terminado de configurar los nombre y tipos es importante pulsar en “Crear tabla” y confirmar la creación para que se genere.

- Hay que crear una regla en AWS IoT para que, cuando se reciba un mensaje en un *topic*, envíe el mensaje a la base de datos creada. Desde la consola de AWS IoT, se accede al menú “Reglas” y de ahí al botón “Crear”:

  - Se da un nombre a la regla.
  - Se recomienda escribir una breve descripción para, cuando se tengan muchas reglas o se dude de qué hace cada una de ellas, tener un resumende lo que hace.
  - En la instrucción de la regla hay que escribir una consulta SQL:

    ```
    SELECT * FROM 'tfg/discoveryIoT/hryt'
    ```

    También se pueden definir las asociaciones de campos de la base de datos con los valores de la publicación `.json`.
  - La versión recomendada de SQL es: 2016-03-23.


Figura 3.5: Consola DynamoDB [57]

– Establecer una acción: Al realizar esta configuración se cambia de ventana, se sale momentáneamente de la creación de la regla pero, cuando se establezca la acción, se volverá a ella. La acción a realizar cuando se reciba un mensaje en el *topic* será: "Insertar un mensaje en una tabla de DynamoDB". A esta acción hay que asignarle un rol con permisos suficientes como para poder añadir registros a una base de datos DynamoDB de AWS. No hay que olvidar pulsar sobre "Añadir acción"

Parece obvio pero,una vez se ha vuelto a la pantalla de creación de la regla, hay que validar todo lo anterior pulsando sobre "Crear regla" y confirmarla.

Estando los mensajes –y por tanto, las mediciones enviadas por el *edge device*– almacenados en una base de datos, estos se pueden enviar a casi todos los servicios de AWS, como por ejemplo, enviar a *AWS Analytics*, *Machine Learning*... para poder extraer estadísticas, predicciones, etc.

# Capítulo 4

# Resultados y conclusiones

## 4.1 Introducción

Debido a la cantidad de partes de diferente naturaleza que componen ente TFG, se ha pasado de puntillas en algunas de ellas, describiendo solo lo suficiente para poder seguir avanzando. Por ejemplo en AWS: es un universo en continua expansión y es muy posible que, desde el momento de escribir la presente memoria hasta el momento de leerla, hayan cambiado opciones, menús, etc. A pesar de ello, se obtiene un sistema que funciona como se deseaba de un principio.

## 4.2 Resultados

Se ha conseguido desarrollar un sistema que, mediante un pequeño hardware externo (la placa *Discovery Kit*) se adquieren datos de humedad y temperatura y se envían a la nube para poder procesarlos con posterioridad.

Los resultados de las adquisiciones se pueden visualizar gráficamente a través de una pequeña y sencilla interfaz escrita en Python 3 que muestra... más bien demuestra que todo el conjunto de configuraciones complejas en AWS (credenciales, *things*, reglas, etc.), análisis y modificación de un ejemplo de Amazon FreeRTOS, configurar desencadenadores para usar otros servicios de AWS... ¡funciona!

Se quedaron muchas cosas en el tintero y otras muchas que se pueden mejorar, pero el objetivo de dar una visión general y práctica de como hacer funcionar *edge devices* con AWS IoT se da por conseguido.

## 4.3 Conclusiones

Siempre es divertido desarrollar un sistema desde el principio. Sobre todo si, al final de todo el proceso, acaba por funcionar como se quería. La contrapartida es que hay que tener tiempo para poder desarrollar algo como el presente trabajo o algo más complejo. Tiempo sobre todo para aprender a desenvolverse en el mundo AWS pues no es nada sencillo y, además, cada poco cambia la interfaz añadiendo más opciones. Tiempo –pero menos– para desarrollar el firmware adaptado a las necesidades que se buscan aunque (y de ahí el "pero menos") con los ejemplos y con los asistentes de configuración se ahorra mucho tiempo que antes (de que existieran) se tenía que emplear.

Comentar que el hardware elegido se escogió porque, a priori, ofrecía unas cualidades concretas necesarias para la realización del mismo, amén de que estaba avalado por Amazon Web Services IoT. También es cierto que fue el más sencillo de encontrar en su momento sin olvidar la economía. Pero todo esto no quiere decir que no haya otros dispositivos/placas de desarrollo que no hagan lo mismo, puede que incluso mejor, pues desde que se comenzó a realizar el trabajo han aparecido multitud de ellos. Pero no es que sólo hayan aparecido muchos, es que cada vez mas tienen más soporte y documentación para programarlos, cosa de lo que quizá sí adolece el kit elegido: documentación buena... no se encontró mucha. Si, mucha publicidad sobre las capacidades y por qué, pero todo presentaciones.

En cuanto a las enrevesadas configuraciones que han de realizarse y que se pueden hacer, bien parece que Amazon no esta por la labor de hacerlas más sencillas. Si bien es cierto que, con un poco de imaginación, se puede vislumbrar que todas y cada una son necesarias para un fin determinado, es como si se buscase más el que se contrate a profesionales formados por ellos mismos para el desarrollo de una aplicación y así todo queda en casa: la plataforma, los dispositivos, los desarrolladores, el desarrollo, el dinero... veremos dónde lleva esto, pero parece que hay Amazon (y crecimiento del mismo) para rato.

# Capítulo 5

# Líneas futuras

Muchos aspectos de esta memoria se pueden desarrollar, tanto a nivel *hardware* como a nivel *software* pues este TFG es una mera introducción a lo que puede hacerse con esa conjunción *hardware* sencillo + "la nube", pero por dar unas pequeñas ideas.

Desde el punto de vista hardware:

- Aumentar la cantidad de medidas: la placa de desarrollo tiene más sensores como el de presión atmosférica, pero quizá lo más "expandible" sea que tiene puertos de entrada y salida con un factor de forma de Arduino, pudiendo acoplarle cualquiera de los *shieds* de expansión de ese gran universo.
- La *Discovery kit* tiene, aparte de la WiFi, otras interfaces para la comunicación inalámbrica, bien se podría virar hacia alguna de ellas y realizar otra comunicación o complementar la vía WiFi
- Debido a la deriva térmica del oscilador del RTC integrado en la placa, se podría integrar una programación de "comprobar la hora" cada cierto tiempo.

Desde el software:

- Emplear otros servicios de AWS, como la inteligencia artificial para obtener predicciones.
- Una característica interesante a investigar es la administración de todo lo referente a AWS a través de una consola CLI (línea de comandos) instalada de forma local con algunas librerías. Ofrece mayor potencia y control pero requiere de un proceso de aprendizaje.
- Probar con otras plataformas en la nube (Azure, Google). Puede que tengan características mejores para según que proyectos y necesidades.
- Hacer que la *Discovery kit* atienda a mensajes publicados por la interfaz gráfica.

# Capítulo 6

# Pliego de condiciones

Las herramientas empleadas para la elaboración completa del proyecto han sido:

- PC compatible con conexión a Internet
- Sistema operativo GNU/Linux Debian 10[35] - Ubuntu 18.04[36]
- ESP32 Devkit C [37]
- STM32L4 Discovery Kit [26]
- Zerynth Studio [29]
- Arduino IDE [33]
- Eclipse Mosquitto [28]
- System Workbench for STM32 de Ac6 [25]

Además de lo anterior, es necesario abrir una cuenta en *Amazon Web Services*. Se trata de una suscripción gratuíta –en el momento de escribir esta memoria– para la actividad que aquí se desarrolla.

# Capítulo 7

# Presupuesto

En este apartado, detallaremos los costes y el presupuesto para la realización del proyecto. Se pormenorizará en cada una de las partidas que dividen en costes cada parte del proyecto.

## 7.1 Coste del material

### 7.1.1 Costes por uso de equipos

| Costes uso Equipos | | | | |
|---|---|---|---|---|
| Equipo | Precio Equipo | Período amortización (meses) | Tiempo de uso (meses) | Coste Amortización |
| PC i7 2.4 GHz 8 GB RAM | 700,00 € | 36 | 6 | 116,67 € |
| | | | **Suma** | **116,67 €** |

Tabla 7.1: Costes uso Equipos

### 7.1.2 Costes compra *hardware*

| Costes *hardware* | | | |
|---|---|---|---|
| Cantidad | Descripción | Precio/ud | Coste Parcial |
| 1 | ST Iot Node | 45,00 € | 45,00 € |
| 1 | ESP32 Dev Kit | 45,00 € | 45,00 € |
| - | Gastos de envío diversos | 40,00 € | 40,00 € |
| - | Componentes discretos | 30,00 € | 30,00 € |
| 1 | PCB microperforada prototipos | 35,00 € | 35,00 € |
| - | Cables y adaptadores | 15,00 € | 15,00 € |
| | | **Suma** | **210,00 €** |

Tabla 7.2: Costes *hardware*

### 7.1.3 Costes *software* y licencias

En cuanto al *software* está la ventaja de que no cuesta dinero, hay que recordar que se trata de Software Libre y puede ser descargado sin tener que pagar una licencia, es decir, tanto el Sistema Operativo como los entornos IDE y algún programa de comunicaciones son de descarga libre de costes.

Si que hay que mencionar que mantener la cuenta de AWS activa no es de coste cero. Al principio, durante seis meses (o un año, dependiendo de condiciones) la capa "*Free tier*" es libre de coste, pero pasado el periodo de prueba, si no se incurre en gastos extras como usar servicios que no estén en la capa gratuita, el coste es de un dólar americano al año (US$ 1). Por aproximación y por no meter valores de divisas se asume que:

**Coste por licencias 1,00 €**

### 7.1.4 Coste total del material

En este apartado sumaremos los costes de uso por equipos, los de compra de dispositivos y los referentes a software.

| Coste Total del Material | |
|---|---|
| Descripción | Coste |
| Coste por uso de equipos | 116,67 € |
| Costes *hardware* | 210,00 € |
| Costes *software* | 1,00 € |
| **Suma** | **327,67 €** |

Tabla 7.3: Coste Total del Material

## 7.2 Coste mano de obra

El proyecto se tarda en realizar unas 120 horas, más la redacción de esta memoria.

| Costes Mano de Obra | | | |
|---|---|---|---|
| Concepto | €/hora | Cantidad horas | Coste |
| Grad. Electrónica de Comunicaciones | 40,00 | 120 | 4.800,00 € |
| Reprografía | - | - | 135,00 € |
| | | **Suma** | **4.935,00 €** |

Tabla 7.4: Costes Mano de Obra

## 7.3 Presupuesto ejecución material

| Presupuesto Ejecución Material | |
|---|---|
| Concepto | Valor |
| Coste Materiales | 327,67 € |
| Coste Mano de Obra | 4.935,00 € |
| **Presupuesto Ejecución** | **5.262,67 €** |

Tabla 7.5: Presupuesto Ejecución Material

## 7.4 Presupuesto contrata

| Presupuesto Contrata | |
|---|---|
| Concepto | Valor |
| Presupuesto Ejecución Material | 5.262,67 € |
| Gastos Generales (17 %) | 894,65 € |
| Beneficio Industrial (6 %) | 315,76 € |
| **Presupuesto Contrata** | **6.473,08 €** |

Tabla 7.6: Presupuesto Contrata

## 7.5 Presupuesto total

| Presupuesto Total | |
|---|---|
| Concepto | Valor |
| Presupuesto Contrata | 6.473,08 € |
| IVA (21 %) | 1.359,35 € |
| **Presupuesto Total** | **7.832,43 €** |

Tabla 7.7: Presupuesto Total

El importe final del proyecto asciende a: **Siete mil ochocientos treinta y dos euros con cuarenta y tres céntimos.**

Alcalá de Henares a 27 de junio de 2021.

Firmado: Sergio Santamaría Quevedo.

Graduado en Ingeniería Electrónica de Comunicaciones.

# Bibliografía


[1] "IBM - Definición de IaaS, PaaS y SaaS," https://www.ibm.com/es-es/cloud/learn/iaas-paas-saas [Último acceso junio/2021].

[2] "¿Qué es IaaS, PaaS e SaaS? Conoce la diferencias," https://www.axarnet.es/blog/saas-paas-iaas/ [Último acceso junio/2021].

[3] D. Bujanda, "Comparación de cloud providers para aplicaciones IoT," https://geoactio.com/2019/02/28/comparacion-de-cloud-providers-para-aplicaciones-iot [Último acceso junio/2021].

[4] A. R. Earls, "AWS vs. Azure and Google: An IoT cloud platform comparison," https://searchaws.techtarget.com/feature/AWS-vs-Azure-and-Google-An-IoT-cloud-platform-comparison [Último acceso junio/2021].

[5] "Listado de dispositivos aprobados por AWS para IoT," https://devices.amazonaws.com/search?page=1 [Último acceso junio/2021].

[6] "Portal AWS," https://aws.amazon.com/es/ [Último acceso junio/2021].

[7] "Servicio AWS IAM," https://docs.aws.amazon.com/es_es/IAM/latest/UserGuide/introduction.html [Último acceso mayo/2021].

[8] "Presentación y portal entrada AWS IoT," https://aws.amazon.com/es/iot/?nc2=h_ql_prod_it [Último acceso jun/2021].

[9] "Presentación y portal entrada AWS IoT Core," https://aws.amazon.com/es/iot-core/?c=i&sec=srv [Último acceso jun/2021].

[10] "Creación de un objeto y certificados en AWS IoT," https://docs.aws.amazon.com/es_es/iot/latest/developerguide/iot-moisture-create-thing.html [Último acceso mayo/2021].

[11] "Creación de políticas en AWS IoT," https://docs.aws.amazon.com/es_es/iot/latest/developerguide/iot-moisture-policy.html [Último acceso mayo/2021].

[12] Varios, Comunicaciones industriales y en Tiempo Real, MÓDULO 2: Sistemas en Tiempo Real: SIS-TEMAS OPERATIVOS PARA TIEMPO REAL (RTOS). JAVA (RTSJ) COMO RTOS. UNED - Departamento de Ingeniería Eléctrica, Electrónica y de Control, ch. 1, http://www.ieec.uned.es/investigacion/Dipseil/PAC/archivos/Informacion_de_referencia_ISE4_2_2.pdf [Último acceso junio/2021].

[13] "Página principal de FreeRTOS TM ," https://www.freertos.org/ [Último acceso junio/2021].

[14] "Página principal del servicio Amazon FreeRTOS," https://aws.amazon.com/es/freertos/ [Último acceso junio/2021].

[15] "AWS Partner Device Catalog: Listado de dispositivos compatibles con Amazon FreeRTOS," https://devices.amazonaws.com/search?page=1&sv=freertos [Último acceso junio/2021].

[16] "AWS IoT Device Tester for FreeRTOS," https://aws.amazon.com/es/freertos/device-tester/ [Último acceso junio/2021].

[17] "Servicio AWS Lambda," https://aws.amazon.com/es/lambda/ [Último acceso junio/2021].

[18] "Bases de datos en AWS," https://aws.amazon.com/es/products/databases/ [Último acceso junio/2021].

[19] "Página de presentación e introducción a DynamoDB en AWS," https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/Introduction.html [Último acceso junio/2021].

[20] "Website de la organización que mantiene el estándar MQTT," http://mqtt.org/ [Último acceso junio/2021].

[21] "ISO/IEC 20922:2016 Information technology ? Message Queuing Telemetry Transport (MQTT) v3.1.1," International Organization for Standardization, June 2016, https://www.iso.org/standard/69466.html [Último acceso junio/2021].

[22] "Página de Wikipedia sobre el formato XML," https://es.wikipedia.org/wiki/Extensible_Markup_Language [Último acceso junio/2021].

[23] "Fuentes de donde se extrajeron los tipos de datos," http://www.json.org/json-es.html; https://es.wikipedia.org/wiki/JSON [Último acceso junio/2021].

[24] "Web principal del lenguaje de programación Python," https://www.python.org/ [Último acceso junio/2021].

[25] "Web principal de System Workbench for STM32," https://www.openstm32.org/System+Workbench+for+STM32 [Último acceso junio/2021].

[26] "Website STM32L4 Discovery kit IoT node," https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html [Último acceso junio/2021].

[27] "Sensor HTS221 en STMicroelectronics," https://www.st.com/en/mems-and-sensors/hts221.html#overview [Último acceso junio/2021].

[28] "Website Eclipse Mosquitto MQTT," https://mosquitto.org// [Último acceso junio/2021].

[29] "Página principal Zerynth - The Middleware for IoT," https://www.zerynth.com/ [Último acceso junio/2021].

[30] "YouTube: Python on ESP32 DevKitC using Zerynth Studio - AWS Connection," https://www.youtube.com/watch?v=IZzZF3DGWkY [Último acceso junio/2021].

[31] Álvaro Benito Herranz, "Desarrollo de aplicaciones para iot con el módulo esp32," Trabajo Final de Grado, Universidad de Alcalá de Henares, 2019.

[32] "Secure IoT with AWS and Hornbill ESP32 using Arduino," https://www.instructables.com/id/Secure-IOT-With-AWS-and-Hornbill-ESP32-Using-Ardui/ [Último acceso junio/2021].

[33] "Página principal de Arduino," https://www.arduino.cc/ [Último acceso junio/2021].

[34] "Tutorial para guardar datos de un dispositivo en DynamoDB en AWS," https://docs.aws.amazon.com/en_en/iot/latest/developerguide/iot-ddb-rule.html [Último acceso abril/2021].

[35] "Website Debian," https://www.debian.org/ [Último acceso junio/2021].

[36] "Website Ubuntu," https://ubuntu.com/ [Último acceso junio/2021].

[37] "Website Espressif ESP32DevKit-C," https://www.espressif.com/en/products/devkits/esp32-devkitc/overview [Último acceso junio/2021].

[38] https://aws.amazon.com/es/iot/?nc2=h_ql_prod_it

[39] https://aws.amazon.com/es/iot/?nc2=h_ql_prod_it

[40] https://aws.amazon.com/es/iot/?nc2=h_ql_prod_it

[41] https://aws.amazon.com/es/iot-core/

[42] https://aws.amazon.com/es/iot-core/?c=i&sec=srv

[43] https://aws.amazon.com/es/iot-core/?c=i&sec=srv

[44] https://www.freertos.org/

[45] By Amazon.com, Inc. The original uploader was Balise42 at English Wikipedia. - Transferred from en.wikipedia to Commons.(Original text : https://aws.amazon.com/architecture/icons/), Public Domain, https://commons.wikimedia.org/w/index.php?

[46] https://aws.amazon.com/es/dynamodb/

[47] By OASIS - https://github.com/mqtt/mqttorg-graphics/blob/master/svg/mqtt-hor.svg, Public Domain, https://commons.wikimedia.org/w/index.php?

[48] De Douglas Crockford - http://clipartist.info//openclipart.org/2011/Sept/September/06-Tuesday/JSON_Logo.svg, Dominio público, https://commons.wikimedia.org/w/index.php?curid=29329762

[49] www.python.org, GPL, https://commons.wikimedia.org/w/index.php?

[50] https://www.eclipse.org/ide/

[51] https://www.ac6-tools.com/content.php/content_SW4MCU/lang_en_GB.xphp

[52] https://www.openstm32.org/HomePage

[53] https://www.espressif.com/en/products/devkits/esp32-devkitc/overview

[54] https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html

[55] Datasheet "UM2153 User manual Discovery kit for IoT node, multi-channel communication with STM32L4" https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html

[56] https://www.electronicwings.com/components/hts221

[57] Extraída de la consola de DynamoDB en AWS de la cuenta de usuario con la que se hizo el presente trabajo.

# Apéndice A

# Código función *Lambda*

```
1  import boto3
2  import json
3  import time
4  from datetime import datetime
5
6  def timestamp():
7          now = datetime.now() #tzMad
8          retorno = []
9          retorno.append(int(time.time()))
10         hora = now.hour
11         minuto = now.minute
12         segundo = now.second
13         segundosDelDia = hora * 3600 + minuto * 60 + segundo
14         nroDiaAnyo = int(now.strftime("%j")) # La %j es para el dIa del anyo [1, 366]
15         retorno.append(nroDiaAnyo * 86400 + segundosDelDia)
16         retorno.append(hora)
17         retorno.append(minuto)
18         retorno.append(segundo)
19         retorno.append(now.day)
20         retorno.append(now.month)
21         retorno.append(now.year)
22         return retorno
23
24 def lambda_handler(event, context):
25     client = boto3.client('iot-data',
26                           aws_access_key_id="<<clave de acceso>>",
27                           aws_secret_access_key="contraseña",
28                           region_name="us-east-2")
29     datetimestamp = timestamp()
```

```
30      menssaje = {"horaUnix" : datetimestamp[0] , "segundoanyo" : datetimestamp[1] ,
31      "horaUTC" : datetimestamp[2] , "minuto" : datetimestamp[3] ,
32      "segundo" : datetimestamp[4] , "dia" : datetimestamp[5] ,
33      "mes" : datetimestamp[6] , "anyo" : datetimestamp[7]}
34
35      response = client.publish(topic='tfg/lambda', qos=1,
36                                payload=json.dumps(menssaje) )
37      return {
38          'statusCode': 100,
39          'body': json.dumps(menssaje)
40      }
```

Listado de código A.1: Función *Lambda*

# Apéndice B

# Código de fuentes `tfgRTC.h/.c`

## B.1 Código de `tfgRTC.h`

```
1  #ifndef FUNCIONES_H_INCLUDED
2  #define FUNCIONES_H_INCLUDED
3
4      #include <stdint.h>
5
6      /// Defines de offset para el calculo de hora UNIX
7      #define OFFSET_horaUnix     12
8      #define OFFSET_segundoanyo  17
9      #define OFFSET_horaUTC      13
10     #define OFFSET_minuto       12
11     #define OFFSET_segundo      13
12     #define OFFSET_dia          9
13     #define OFFSET_mes          9
14     #define OFFSET_anyo         10
15
16     /// Estructura timestamp
17     typedef struct {    /// tStruMsjJSON
18         uint32_t    horaUNIX;
19         uint32_t    segundoanyo;
20         uint8_t     horaUTC;
21         uint8_t     minuto;
```

```
22          uint8_t     segundo;
23          uint8_t     dia;
24          uint8_t     mes;
25          uint16_t    anyo;
26      } tStruMsjJSON;
27
28      /// Estructura para RTC
29      typedef struct RtcDatetime {
30          uint16_t year;    /// Anyo:    desde 1970 hasta 2099
31          uint16_t month;   /// Mes:     desde    1 hasta 12
32          uint16_t day;     /// Dia:     desde    1 hasta 31 (dependiendo del mes)
33          uint16_t hour;    /// Hora:    desde    0 hasta 23
34          uint16_t minute;  /// Minuto:  desde    0 hasta 59
35          uint8_t second;   /// Segundo: desde    0 hasta 59
36      } rtc_datetime_t;
37
38      /// Declaracion de funciones
39      uint8_t analizaJSON(uint8_t *, tStruMsjJSON *);
40      void convierteStruJSONaHoraUNIX(tStruMsjJSON * datetime, uint32_t * seconds);
41
42  #endif // FUNCIONES_H_INCLUDED
```

Listado de código B.1: `tfgRTC.h`

## B.2 Código de **tfgRTC.c**

```
1   #include <stdint.h>
2   #include <string.h>
3   #include <stdlib.h>
4
5   #include "main.h"
6   #include "tfgRTC.h"
7
8   #define DIAS_EN_UN_ANYO         (365U)
9   #define SEGUNDOS_EN_UN_DIA      (86400U)
10  #define SEGUNDOS_EN_UNA_HORA    (3600U)
11  #define SEGUNDOS_EN_UN_MIN      (60U)
12
13  /// Número de días desde el inicio del año no bisiesto
14  static const uint16_t DIAS_D_MESES[] = {0U, 0U, 31U, 59U, 90U, 120U, 151U, 181U,
15      212U, 243U, 273U, 304U, 334U};
16
17  /*************************************************************************
18   * Función que analiza el array recibido y devuelve los valores
19   *     en la estructura que es pasada por referencia
20   * Parámetros:  puntero a la cadena a analizar ;; puntero a la estructura
21   *     a la que devolver los valores
22   * Retorno: estado de la conversión
23   *************************************************************************/
24  uint8_t analizaJSON(uint8_t *mensaje, tStruMsjJSON *valoresJSON) {
25      uint8_t     resultado = 1, // 1 = OK
26                  indice, indiceAux;
27      char        *pEtiqueta;
28      char        cadenaAux[20];
29
30      /// Primer valor/etiqueta: "horaUnix"
```

```
31      pEtiqueta = NULL;
32      pEtiqueta = strstr( (char *) mensaje, "\"horaUnix\": ");     /// El puntero apuntará a
            '"' .OR. NULL
33      if (pEtiqueta != NULL) {
34          indiceAux = 0;
35          indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_horaUnix;
36          while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
37              cadenaAux[indiceAux++] = (char) mensaje[indice++];
38          }
39          cadenaAux[indiceAux] = 0;
40          valoresJSON->horaUNIX = atoi(cadenaAux);
41      }
42      else
43          resultado = 0;
44
45      if (resultado == 1) {
46          /// Segundo valor/etiqueta: "segundoanyo"
47          pEtiqueta = NULL;
48          pEtiqueta = strstr( (char *) mensaje, ", \"segundoanyo\": ");    /// El puntero
                apuntará a ',' .OR. NULL
49          if (pEtiqueta != NULL) {
50              indiceAux = 0;
51              indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_segundoanyo;
52              while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
53                  cadenaAux[indiceAux++] = (char) mensaje[indice++];
54              }
55              cadenaAux[indiceAux] = 0;
56              valoresJSON->segundoanyo = atoi(cadenaAux);
57          }
58          else
59              resultado = 0;
60      }
61
62      if (resultado == 1) {
63          /// Tercer valor/etiqueta: "horaUTC"
64          pEtiqueta = NULL;
65          pEtiqueta = strstr( (char *) mensaje, ", \"horaUTC\": ");    /// El puntero apuntará
                a ',' .OR. NULL
66          if (pEtiqueta != NULL) {
67              indiceAux = 0;
68              indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_horaUTC;
69              while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
70                  cadenaAux[indiceAux++] = (char) mensaje[indice++];
71              }
72              cadenaAux[indiceAux] = 0;
73              valoresJSON->horaUTC = atoi(cadenaAux);
74          }
75          else
76              resultado = 0;
77      }
78
79      if (resultado == 1) {
80          /// Cuarto valor/etiqueta: "minuto"
81          pEtiqueta = NULL;
82          pEtiqueta = strstr( (char *) mensaje, ", \"minuto\": ");     /// El puntero apuntará
                a ',' .OR. NULL
83          if (pEtiqueta != NULL) {
84              indiceAux = 0;
85              indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_minuto;
```

```
86              while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
87                  cadenaAux[indiceAux++] = (char) mensaje[indice++];
88              }
89              cadenaAux[indiceAux] = 0;
90              valoresJSON->minuto = atoi(cadenaAux);
91          }
92          else
93              resultado = 0;
94      }
95
96      if (resultado == 1) {
97          /// Quinto valor/etiqueta: "segundo"
98          pEtiqueta = NULL;
99          pEtiqueta = strstr( (char *) mensaje, ", \"segundo\": ");   /// El puntero apuntará
                a ',' .OR. NULL
100         if (pEtiqueta != NULL) {
101             indiceAux = 0;
102             indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_segundo;
103             while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
104                 cadenaAux[indiceAux++] = (char) mensaje[indice++];
105             }
106             cadenaAux[indiceAux] = 0;
107             valoresJSON->segundo = atoi(cadenaAux);
108         }
109         else
110             resultado = 0;
111     }
112
113     if (resultado == 1) {
114         /// Sexto valor/etiqueta: "dia"
115         pEtiqueta = NULL;
116         pEtiqueta = strstr( (char *) mensaje, ", \"dia\": ");   /// El puntero apuntará a
                ',' .OR. NULL
117         if (pEtiqueta != NULL) {
118             indiceAux = 0;
119             indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_dia;
120             while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
121                 cadenaAux[indiceAux++] = (char) mensaje[indice++];
122             }
123             cadenaAux[indiceAux] = 0;
124             valoresJSON->dia = atoi(cadenaAux);
125         }
126         else
127             resultado = 0;
128     }
129
130     if (resultado == 1) {
131         /// Séptimo valor/etiqueta: "mes"
132         pEtiqueta = NULL;
133         pEtiqueta = strstr( (char *) mensaje, ", \"mes\": ");   /// El puntero apuntará a
                ',' .OR. NULL
134         if (pEtiqueta != NULL) {
135             indiceAux = 0;
136             indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_mes;
137             while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
138                 cadenaAux[indiceAux++] = (char) mensaje[indice++];
139             }
140             cadenaAux[indiceAux] = 0;
141             valoresJSON->mes = atoi(cadenaAux);
```

```
142         }
143         else
144             resultado = 0;
145     }
146 
147     if (resultado == 1) {
148         /// Octavo valor/etiqueta: "anyo"
149         pEtiqueta = NULL;
150         pEtiqueta = strstr( (char *) mensaje, ", \"anyo\": ");  /// El puntero apuntará a
                ',' .OR. NULL
151         if (pEtiqueta != NULL) {
152             indiceAux = 0;
153             indice = (uint8_t) (pEtiqueta - (char *) mensaje) + OFFSET_anyo;
154             while (mensaje[indice] >= 0x30 && mensaje[indice] <= 0x39) {
155                 cadenaAux[indiceAux++] = (char) mensaje[indice++];
156             }
157             cadenaAux[indiceAux] = 0;
158             valoresJSON->anyo = atoi(cadenaAux);
159         }
160         else
161             resultado = 0;
162     }
163 
164     return resultado;
165 }
166 
167 /*****************************************************************************
168  * Función que convierte un timestamp en segundos UNIX
169  *
170  * Parámetros:  puntero a estructura timestamp ;; puntero a la variable donde
171  *      guardar los segundos UNIX
172  *
173  *****************************************************************************/
174 void convierteStruJSONaHoraUNIX(tStruMsjJSON * datetime, uint32_t * segundos) {
175     /// Calulamos la cantidad de días desde 1970 hasta el año dado
176     *segundos = (datetime->anyo - 1970U) * DIAS_EN_UN_ANYO;
177     /// Se añaden los días de años bisiestos
178     *segundos += ((datetime->anyo / 4) - (1970U / 4));
179     /// Se suman los días hasta el mes dado
180     *segundos += DIAS_D_MESES[datetime->mes];
181     /// Se suman los días del mes dado. Se quitan los segundos del día actual ya
182     /// que se verán reflejados en el campo de horas y segundos
183     *segundos += (datetime->dia - 1);
184     /// Para años bisiestos, si el mes es menor o igual a febrero, hay que
185     /// disminuir el contador de días
186     if ((!(datetime->anyo & 3U)) && (datetime->mes <= 2U)) {
187         (*segundos)--;
188     }
189 
190     *segundos = ((*segundos) * SEGUNDOS_EN_UN_DIA) + (datetime->horaUTC *
            SEGUNDOS_EN_UNA_HORA) +
191             (datetime->minuto * SEGUNDOS_EN_UN_MIN) + datetime->segundo;
192 }
```

Listado de código B.2: tfgRTC.c

# Apéndice C

# Código de la interfaz gráfica en Python

```
#!/usr/bin/python3

"""
Created on May 2021
@author: Zegio
"""
import threading
import tkinter as tk
import matplotlib.pyplot as plt
import matplotlib.dates as md
import time
import argparse
import sys
import json
import math     ## para modo TEST
try:
    import Queue as queue
except ImportError:
    # Python 3
    import queue
from tkinter import messagebox
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from datetime import datetime
from awscrt import io, mqtt, auth, http
from awsiot import mqtt_connection_builder
plt.style.use('ggplot')

################################################################################
cola_fifo = queue.Queue(12)
```

```
30
31  """
32  Variables globales de la interfaz grafica
33  """
34  MUESTRAS = 200
35  #-- Variable para indicar al thread que termine
36  fin = False
37
38  """
39  Variables globales de la parte AWS MQTT
40  """
41  salir = False
42  primerMensajeJson = False
43  mensajejson_anterior = []
44  received_count = 0
45  received_all_event = threading.Event()
46
47  """
48  Variables globales del modo test
49  """
50  indice_test = 1623996000
51  instanteFinal = datetime.now()
52  instanteInicial = datetime.now()
53  un_decimo_2pi = math.pi / (MUESTRAS / 2) #0,62831853
54  contaSen = 0
55  esperandoFinHilo = False
56
57  ##############################################################################
58  """
59  Hilo para el modo test
60  """
61  def hilo_fuera_de_tk():
62      global fin
63      global esperandoFinHilo
64      global indice_test
65      global instanteFinal
66      global instanteInicial
67      global un_decimo_2pi
68      global contaSen
69      while not fin:
70          if esperandoFinHilo:
71              print ('hilo_fin = ', fin)
72          instanteFinal = datetime.now()
73          tiempo = instanteFinal - instanteInicial # Devuelve un objeto timedelta
74          segundos = tiempo.microseconds
75          if segundos >= 250000:
76              contaSen += 1
77
78              cola_fifo.put(indice_test)
79              indice_test += 1
80
81              hume_aleatoria = (35 * math.sin(un_decimo_2pi * (contaSen % MUESTRAS) ) ) + 55
82              cola_fifo.put(hume_aleatoria)
83
84              temp_aleatoria = (23 * math.cos(un_decimo_2pi * (contaSen % MUESTRAS) ) ) + 18
85              cola_fifo.put(temp_aleatoria)
86
87              instanteInicial = datetime.now()
88
89  ##############################################################################
90  """
91  Funciones de la parte AWS MQTT
92  """
93  def config_AWS_MQTT():
94      parser = argparse.ArgumentParser(description = "Send and receive messages through and MQTT connection.")
95
96      parser.add_argument('--endpoint',
97                          default="aivjcbhlx43wb-ats.iot.us-east-2.amazonaws.com",
98                          help="Your AWS IoT custom endpoint, not including a port." +
99                          " Ex: \"abcd123456wxyz-ats.iot.us-east-1.amazonaws.com\"")
```

```
100     parser.add_argument('--cert',
101                         default="certificados/7bbf3137ed-certificate.pem.crt",
102                         help="File path to your client certificate, in PEM format.")
103     parser.add_argument('--key', default="certificados/7bbf3137ed-private.pem.key",
104                         help="File path to your private key, in PEM format.")
105     parser.add_argument('--root-ca', default="certificados/CA_x509_RSA_2048b.pem",
106                         help="File path to root certificate authority, in PEM " +
107                         "format. Necessary if MQTT server uses a certificate " +
108                         "that's not already in your trust store.")
109     parser.add_argument('--client-id', default="pyTest",
110                         help="Client ID for MQTT connection.")
111     parser.add_argument('--topic', default="tfg/#",
112                         help="Topic to subscribe to, and publish messages to.")
113     parser.add_argument('--message', default="Hola Mundo",
114                         help="Message to publish. Specify empty string to" +
115                         " publish nothing.")
116     parser.add_argument('--count', default=10, type=int,
117                         help="Number of messages to publish/receive before exiting. " +
118                         "Specify 0 to run forever.")
119     parser.add_argument('--use-websocket', default=False, action='store_true',
120                         help="To use a websocket instead of raw mqtt. If you " +
121                         "specify this option you must specify a region for signing, " +
122                         "you can also enable proxy mode.")
123     parser.add_argument('--signing-region', default='us-east-1', help="If you specify " +
124                         "--use-web-socket, this " +
125                         "is the region that will be used for computing the Sigv4 signature")
126     parser.add_argument('--proxy-host', help="Hostname for proxy to connect to. Note: " +
127                         "if you use this feature, you will likely need to set " +
128                         "--root-ca to the ca for your proxy.")
129     parser.add_argument('--proxy-port', type=int, default=8080,
130                         help="Port for proxy to connect to.")
131     parser.add_argument('--verbosity', choices=[x.name for x in io.LogLevel],
132                         default=io.LogLevel.NoLogs.name, help='Logging level')
133
134     # Se usan globales para simplificar el codigo
135     argumentos = parser.parse_args()
136
137     return argumentos
138
139 # 'Callback' cuando se pierde la conexion (accidentalmente)
140 def conexion_interrumpida(connection, error, **kwargs):
141     print("ConexiOn interrumpida. error: {}".format(error))
142
143 # 'Callback' cuando una conexion es reestablecida
144 def conexion_reestablecida(connection, return_code, session_present, **kwargs):
145     print("ConexiOn retomada. return_code: {} session_present: {}".format(return_code, session_present))
146
147     if return_code == mqtt.ConnectReturnCode.ACCEPTED and not session_present:
148         print("La sesiOn no persistiO. Resuscribiendo al topic existente...")
149         resubscribe_future, _ = connection.resubscribe_existing_topics()
150
151         # No se puede sincronizar, se espera para re-suscribirse
152         resubscribe_future.add_done_callback(reconexion_completada)
153
154 def reconexion_completada(resubscribe_future):
155         resubscribe_results = resubscribe_future.result()
156         print("Resultados de resuscripciOn: {}".format(resubscribe_results))
157
158         for topic, qos in resubscribe_results['topics']:
159             if qos is None:
160                 sys.exit("El servidor rechazO suscribirse al topic: {}".format(topic))
161
162 # 'Callback' cuando el topic suscrito recibe un mensaje
163 def mensaje_mqtt_recibido(topic, payload, **kwargs):
164     global salir, primerMensajeJson, mensajejson_anterior
165     global args
166     global received_count
167     global received_all_event
168     # el argumento payload es <class 'bytes'> asi que, para posteriores tratamientos:
169     mensajejson = payload.decode("utf-8")
```

```
170
171     if primerMensajeJson or mensajejson_anterior != mensajejson:
172         primerMensajeJson = False
173         mensajejson_anterior = mensajejson
174         jsonDecodificado = json.loads(mensajejson)
175         try:
176             indiceMsg = jsonDecodificado["ind"]
177             humedadMsg = jsonDecodificado["humedad"]
178             temperaturaMsg = jsonDecodificado["temperatura"]
179             if indiceMsg != -1:
180                 cola_fifo.put(indiceMsg)
181                 cola_fifo.put(humedadMsg)
182                 cola_fifo.put(temperaturaMsg)
183             else:
184                 salir = True
185                 print ("Detectado -1 para salir.")
186
187         except KeyError:    # error que se da cuando no hay entrada en el diccionario
188             print ('--------------------------------------------------------------------------')
189             print ("--  Mensaje NO reconocido  --")
190             print ("------------------------------")
191             print ("Topic:   ", topic)
192             print ("Payload: ", mensajejson)
193             print ('--------------------------------------------------------------------------')
194
195         received_count += 1
196         if received_count == args.count:
197             received_all_event.set()
198         else:
199             received_all_event.clear()
200
201 def hilo_AWS_MQTT():
202     global args
203     global received_count
204     args = config_AWS_MQTT()
205
206     io.init_logging(getattr(io.LogLevel, args.verbosity), 'stderr')
207
208     event_loop_group = io.EventLoopGroup(1)
209     host_resolver    = io.DefaultHostResolver(event_loop_group)
210     client_bootstrap = io.ClientBootstrap(event_loop_group, host_resolver)
211
212     if args.use_websocket == True:
213         proxy_options = None
214         if (args.proxy_host):
215             proxy_options = http.HttpProxyOptions(host_name = args.proxy_host, port = args.proxy_port)
216         credentials_provider = auth.AwsCredentialsProvider.new_default_chain(client_bootstrap)
217         mqtt_connection = mqtt_connection_builder.websockets_with_default_aws_signing(
218             endpoint                 = args.endpoint,
219             client_bootstrap         = client_bootstrap,
220             region                   = args.signing_region,
221             credentials_provider     = credentials_provider,
222             websocket_proxy_options  = proxy_options,
223             ca_filepath              = args.root_ca,
224             on_connection_interrupted = conexion_interrumpida,
225             on_connection_resumed    = conexion_reestablecida,
226             client_id                = args.client_id,
227             clean_session            = False,
228             keep_alive_secs          = 6)
229     else:
230         mqtt_connection = mqtt_connection_builder.mtls_from_path(
231             endpoint                 = args.endpoint,
232             cert_filepath            = args.cert,
233             pri_key_filepath         = args.key,
234             client_bootstrap         = client_bootstrap,
235             ca_filepath              = args.root_ca,
236             on_connection_interrupted = conexion_interrumpida,
237             on_connection_resumed    = conexion_reestablecida,
238             client_id                = args.client_id,
239             clean_session            = False,
```

```
240             keep_alive_secs           = 6)
241
242     print("Conectando a '{}' con clientID '{}'...".format(args.endpoint, args.client_id))
243     connect_future = mqtt_connection.connect()
244
245     # Future.result() waits until a result is available
246     connect_future.result()
247     print("Conectado.")
248
249     # Subscribe
250     print("Suscribiendose al topic '{}'...".format("tfg/#")) #tfg/discoveryIoT/hryt"))  #format(args.topic))
251     subscribe_future, packet_id = mqtt_connection.subscribe(
252                                   #topic    = args.topic,
253                                   topic    = "tfg/#",
254                                   #topic    = "tfg/discoveryIoT/hryt",
255                                   qos      = mqtt.QoS.AT_LEAST_ONCE,
256                                   callback = mensaje_mqtt_recibido)
257
258     subscribe_result = subscribe_future.result()
259     print("Subscribed with {}".format(str(subscribe_result['qos'])))
260
261     # Publish message to server desired number of times.
262     # This step is skipped if message is blank.
263     # This step loops forever if count was set to 0.
264
265     #pg.QtGui.QApplication.exec_()
266
267     args.count = 0
268     if args.message:
269         if args.count == 0:
270             print ("Pendiente de mensajes hasta matar programa o recibir 'ind': -1")
271         else:
272             print ("Sending {} message(s)".format(args.count))
273
274         while args.count == 0 and not salir:
275             time.sleep(0.5)
276
277     print("{} mensaje(s) recibido(s).".format(received_count))
278
279     # Desconectando
280     print("Desconectando del broker...", end="")
281     disconnect_future = mqtt_connection.disconnect()
282     disconnect_future.result()
283     print(" desconectado")
284
285 ##############################################################################
286 """
287 Clase de la interfaz grafica
288 """
289 class InterfazGraficaTFG():
290     global cola_fifo
291
292     def __init__(self):
293         self.MIN_ESC_TEMP = -10
294         self.MAX_ESC_TEMP = 50
295         self.MIN_ESC_HUME = 5
296         self.MAX_ESC_HUME = 95
297         self.texto_info = ''
298         self.nroValoresAmostrar = MUESTRAS
299         self.conValoresIni = False
300         self.habilitadaSalidaTexto = True
301
302         #------------------
303         #------ Raiz ------
304         #------------------
305         self.raiz = tk.Tk()
306         self.raiz.geometry('900x675')
307         self.raiz.title("TFG: AWS + Tkinter + Matplotlib")
308         #------------------
309         #----- Frames -----
```

```
310         #------------------
311         self.frame_izq = tk.Frame(self.raiz)
312         self.frame_dch = tk.Frame(self.raiz, bg = '#C0C0C0', bd = 1.5)
313         self.frame_inf = tk.Frame(self.raiz)
314         self.frame_izq.place(relx = 0.02, rely = 0.02, relwidth = 0.27, relheight = 0.77)
315         self.frame_dch.place(relx = 0.31, rely = 0.02, relwidth = 0.67, relheight = 0.77)
316         self.frame_inf.place(relx = 0.02, rely = 0.81, relwidth = 0.96, relheight = 0.17)
317         #------------------
318         #- Elementos base -
319         #------------------
320         self.B0     = tk.Button(self.frame_izq, text = "Boton a programar",   command = self.B0f,
321                                                                                state = tk.DISABLED)
322         self.B1     = tk.Button(self.frame_izq, text = "Actualizar grafica",  command = self.dibujaGrafica)
323         self.B2     = tk.Button(self.frame_izq, text = "Des/Habilita salida", command = self.B2f)
324         self.Acerca = tk.Button(self.frame_izq, text = "Acerca de...",        command = self.B_acerca)
325         self.tinfo  = tk.Text(self.frame_inf)#, width = 15, height = 3)
326
327         alturaRel = 0.19
328         self.B0.place    (relheight=alturaRel, relwidth=1)
329         self.B1.place    (rely =   (0.1 + alturaRel*0.54), relheight = alturaRel, relwidth = 1)
330         self.B2.place    (rely = 2*(0.1 + alturaRel*0.54), relheight = alturaRel, relwidth = 1)
331         self.Acerca.place(rely = 4*(0.1 + alturaRel*0.54), relheight = alturaRel, relwidth = 1)
332         self.tinfo.place (relheight = 1, relwidth = 1)
333
334         #------------------
335         #- Agregar figura -
336         #------------------
337         self.formatoDistDia = md.DateFormatter('%Y-%m-%d %H:%M:%S')
338         self.inclEjeX = 20
339         self.formatoMismDia = md.DateFormatter('%H:%M:%S')
340         self.etiquetaEjeX = 'hora UNIX'
341
342         self.figura = plt.Figure()
343
344         self.subplt_t = self.figura.add_subplot(111, label = "1")
345         self.subplt_h = self.figura.add_subplot(111, label = "2", frame_on = False)
346         self.figura.subplots_adjust(bottom = 0.2)
347
348         self.subplt_t.clear()
349         self.subplt_h.clear()
350
351         self.subplt_t.grid(False)
352         self.subplt_h.grid(False)
353
354         self.subplt_t.xaxis.set_major_formatter( self.formatoDistDia )
355
356         self.subplt_t.set_ylim(self.MIN_ESC_TEMP, self.MAX_ESC_TEMP)
357         self.subplt_h.set_ylim(self.MIN_ESC_HUME, self.MAX_ESC_HUME)
358
359         self.subplt_t.set_xlabel(self.etiquetaEjeX, color="C0")
360         self.subplt_t.set_ylabel("temperatura", color="C0")
361         self.subplt_t.tick_params(axis = 'x', labelrotation = self.inclEjeX)
362         self.subplt_t.tick_params(axis = 'y', colors = "C0")
363
364         self.subplt_h.yaxis.tick_right()
365         self.subplt_h.set_ylabel('HR', color = "C1")
366         self.subplt_h.yaxis.set_label_position('right')
367         self.subplt_h.tick_params(axis = 'y', colors = "C1")
368         self.subplt_h.get_xaxis().set_visible(False)
369
370         self.graficoTk = FigureCanvasTkAgg(self.figura, self.frame_dch)
371         self.graficoTk.get_tk_widget().pack(side = tk.LEFT, fill = tk.BOTH, expand = 1)
372
373         self.valores_x  = [0]
374         self.valores_y1 = [0]
375         self.valores_y2 = [0]
376
377         #--------------------
378         #- Funcion ciclica -
379         #--------------------
```

```
380         self.raiz.after(2000, self.func_ciclica)
381         #-------------------
382 
383         self.raiz.mainloop()
384 
385     ########################################################################
386     def B0f(self):
387         pass
388 
389     def dibujaGrafica(self):
390         self.subplt_t.clear()
391         self.subplt_h.clear()
392 
393         self.subplt_t.grid(False)
394         self.subplt_h.grid(False)
395 
396         self.subplt_t.set_ylim(self.MIN_ESC_TEMP, self.MAX_ESC_TEMP)
397         self.subplt_h.set_ylim(self.MIN_ESC_HUME, self.MAX_ESC_HUME)
398 
399         dat = [datetime.fromtimestamp(marca) for marca in self.valores_x]
400         dateAnums = md.date2num(dat)
401         if (self.valores_x[-1] - self.valores_x[0]) > 86400:
402             self.subplt_t.xaxis.set_major_formatter( self.formatoDistDia )
403             self.etiquetaEjeX = 'hora UNIX'
404         else:
405             self.subplt_t.xaxis.set_major_formatter( self.formatoMismDia )
406             dia_temp = datetime.fromtimestamp(self.valores_x[0])
407             self.etiquetaEjeX = 'hora UNIX ({}/{}/{})'.format(dia_temp.year, dia_temp.month, dia_temp.day)
408 
409         self.subplt_t.plot(dateAnums, self.valores_y1, color="C0")
410         self.subplt_t.set_xlabel(self.etiquetaEjeX, color="C0")
411         self.subplt_t.set_ylabel("temperatura", color="C0")
412         self.subplt_t.tick_params(axis = 'x', labelrotation = self.inclEjeX)
413         self.subplt_t.tick_params(axis = 'y', colors = "C0")
414 
415         self.subplt_h.plot(dateAnums, self.valores_y2, color="C1")
416         self.subplt_h.yaxis.tick_right()
417         self.subplt_h.set_ylabel('HR', color="C1")
418         self.subplt_h.yaxis.set_label_position('right')
419         self.subplt_h.tick_params(axis='y', colors="C1")
420 
421         self.graficoTk.draw()
422 
423     def B2f(self):
424         if self.habilitadaSalidaTexto:
425             self.printf('Deshabilitando salida de texto.')
426             self.tinfo.config(state = 'disable')
427             self.habilitadaSalidaTexto = False
428         else:
429             self.tinfo.config(state = 'normal')
430             self.printf('Habilitando salida de texto.')
431             self.habilitadaSalidaTexto = True
432 
433     def B_acerca(self):
434         messagebox.showinfo('Acerca de...', 'Interfaz grafica\npara el TFG\nSub-Pub MQTT\n\nby Zegio')
435 
436     def printf(self, cadena):
437         self.texto_info = cadena + '\n'
438         self.tinfo.insert("1.0", self.texto_info)
439 
440     def func_ciclica(self):
441         try:
442             indi = cola_fifo.get()
443             hume = cola_fifo.get()
444             temp = cola_fifo.get()
445             if self.conValoresIni:
446                 self.printf('Valores de: horaUTC = {} ;; temp = {:>2.2f}Â°C ;; hume = {:>2.2f}%'
447                             .format(datetime.utcfromtimestamp(indi).strftime('%Y-%m-%d %H:%M:%S'), temp, hume))
448                 self.valores_x.append(indi)
449                 self.valores_y1.append(temp)
```

```
450                 self.valores_y2.append(hume)
451
452                 if len(self.valores_x) > self.nroValoresAmostrar:
453                     self.valores_x  = self.valores_x[1:]
454                     self.valores_y1 = self.valores_y1[1:]
455                     self.valores_y2 = self.valores_y2[1:]
456             else:
457                 self.conValoresIni = True
458                 self.valores_x[0]  = indi
459                 self.valores_y1[0] = temp
460                 self.valores_y2[0] = hume
461
462             self.dibujaGrafica()
463         except:
464             pass
465
466         # Volvemos a decir que se ejecute dentro de 4000 milisegundos
467         self.raiz.after(500, self.func_ciclica)
468
469 ###############################################################################
470 def main():
471     global fin
472
473     hilo_aws= threading.Thread(target = hilo_AWS_MQTT)
474     hilo_aws.start()
475
476     mi_app = InterfazGraficaTFG()
477     print ('\nFinalizada interfaz grafica.')
478
479     fin = True
480
481     print ('\nEsperando que termine el hilo.')
482     hilo_aws.join()
483     time.sleep(1)
484
485     print('\nHilo terminado.')
486
487     return 0
488
489 def main_test():
490     global fin
491     global instanteInicial
492     global esperandoFinHilo
493
494     print ('\n---------------')
495     print ('-- Modo TEST --')
496     print ('---------------')
497
498     #-- Lanzar el hilo (como daemon) que lee del puerto serie
499     hilo_ftk= threading.Thread(target = hilo_fuera_de_tk)
500     hilo_ftk.start()
501     instanteInicial = datetime.now()
502
503     mi_app = InterfazGraficaTFG()
504     print ('\nFinalizada interfaz grafica.')
505
506     fin = True
507     esperandoFinHilo = True
508
509     print ('\nEsperando que termine el hilo.')
510     hilo_ftk.join()
511
512     time.sleep(1)
513
514     print('\nHilo terminado.')
515
516     return 0
517
518
519 ###############################################################################
```

```
520 if __name__ == '__main__':
521
522     if len(sys.argv) > 1:
523         if sys.argv[1] == 'test':
524             main_test()
525         else:
526             print ("Argumentos incorrectos.")
527     else:
528         main()
```

Listado de código C.1: Interfaz gráfica en Python

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá